phthon-seekho

Ather Abbas

Nov 11, 2024

CONTENTS

CHAPTER

ONE

FOREWORD

A self-paced beginner-level book for python-learners.

1.1 Background

I must confess that I do not consider myself an expert capable of writing an entire book on Python. As I embarked on my journey to learn Python through online resources, I encountered a wealth of materials readily available on the internet. Whenever I stumbled upon a particularly insightful example that helped clarify a concept, I copied and saved that code snippet in a centralized location. Slowly, this copied material grew larger, and I started organizing it so that other people could also benefit from the code snippets which I found useful and which were scattered here and there. Moreover, as I was learning python, I discovered that hands-on practice is the most effective approach. However, there are very few resources which offer their code material for download. This led to the creation of the resource that you have in your hand or on your screen right now!

1.2 Purpose

The purpose of this book is to provide prospective learners a self-taught material with executable notebooks. You can consider this book as a collection of practical code exercises when the (python) infrastructure is available. However, the pdf version of it can also be used as an ebook. The book is more closely related to the python section of the tutorialspoint website. However, this book covers many topics which are not discussed there. More importantly, the user can download the code examples here as either Jupyter notebook or as .py files. Therefore, the user can access the complete material which is more helpful for practical purposes.

1.3 Who should read this book

This book is intended for absolute beginners, those who have no background in python. A very little knowledge of programming can be helpful for the readers. The advanced python users may find advanced topics within each chapter in this book useful.

1.4 Installation

If you have internet connection, you can download the ipython notebook of a lesson and upload it on any cloud servers such as colab , binder or studiolab .

For offline users; first install python, then install all the libraries mentioned in requirements file as well as jupyter. Then the user can run any jupyter notebook which again can be downloaded from website .

1.5 How to read this book?

This book consists of Jupyter notebooks. Each Jupyter notebook consists of multiple cells. Each cell consists of code snippets. The best way to get the maximum out of this book is to follow following two tips

1) Predict the output from a cell before running the cell, then run the cell and see whether your prediction matches with the output of the cell or not!
2) Change the contents of the cell and predict the output and then run the cell and compare your prediction with actual output.
3) Answer the questions asked in the lessons.

1.6 Conventions

Since this is an executable book, which means all the code in this book is executed during the creation of the book. There is some code in the book which throws errors and has been commented out. The code which raises errors is only for demonstration purpose and does not represent a bug in the book. The readers are explicitly advised to uncomment those lines and run the code to see and understand the error. For example, the following code illustrates the working of `TypeError`. However, the code which raises the `TypeError` is commented out. When the user uncomments the line and runs it, he/she can see the error.

```
>>> a = 2

>>> # uncomment following line
>>> # a + 'a'  # -> TypeError, we can not add integer and string types
```

1.7 Note

This is a living book which is currently being maintained actively on its GitHub repository. You can expect improvement in the book with time.

CHAPTER

TWO

CHAPTERS

2.1 1. basics

Tutorials concerning python basics

2.2 2. builtin modules

This chapter describes some useful built-in modules/packages in python.

2.3 3. oop

Tutorials concerning object oriented programming

2.4 4. numpy

Tutorials concerning numpy library .

2.5 5. pandas

Tutorials concerning pandas library

2.6 6. plotting

This chapter contains lessons regarding plotting.

2.7 7. Advanced

2.7.1 1. basics

Tutorials concerning python basics

1.1 variables

This lesson introduces data types and variables in python.

A variable is a way to link some data to a memory location. The memory here, does not mean the storage such as hard drive or USB etc. rather memory such as RAM. The memory size which is allocated for a variable depends on the *kind* and the amount of data linked to that variable. For example, a variable consisting 10 integers will hold less memory as compared to a variable consisting of a million integers. Similarly, a variable holding an integer (e.g. 92) will have different amount/size of memory as compared to a variable holding a string say `Ali`. When we define a variable in python, the python checks the type of the variable and allocates some memory for that variable.

```
a = 12
```

So what has been done above is that a variable named `a` is assigned a value of 12. Behind the scenes python created an `object` and the variable name `a` is a reference for that object.

```
a = 14
```

When the variable `a` is redefined, it means the location of memory which was holding the value 12 before, now holds 14. This means we have changed the contents of memory.

```
a = a + 12
print(a)
```

```
26
```

The `print` is a (builtin) function in python which we can use to *print the value of a variable*. This is not always true but more about it will come in *1.11 print* and *3.16 magic methods*. The kind of object python creates, depends upon the type of data. We can check the *type* of a variable by using the command/function `type(VariableName)`

```
print(type(a))
```

```
<class 'int'>
```

A `function` is a different creature in python. We will cover more about it later in *1.12 functions* . At this point, just keep in mind that when some object is a function, we can *call* it by appending paranthesis `()` after its name Above we have called the `print` function. Don't worry if you don't understand the meaning of calling a function at this point.

The function `type` is **the most important function in python**. Whenver you don't know about some object in python, the first thing you should do is to check its `type` using `type(object)`

```
b = 30.0
print(type(b))
```

```
<class 'float'>
```

Question

Both a and b contained the value *thirty*, then why they had different types?

```
c = a + b
print(c, type(c))
```

```
56.0 <class 'float'>
```

```
d = a + a
print(d, type(d))
```

```
52 <class 'int'>
```

It is important to note that the python is able to change the type of new variable based on the kind of value assigned to it. If a float value is assigned to a variable, python will change the type of this variable to float.

Another significant thing is that, we can assign any type of data to a variable. For example, we can assign `int` to variable a, later we can assign `float` to the variable a and then we can assign a completely different type like `string` to the variable a.

This is a blessing (in terms of ease of use) as well as a curse (in terms of its slow speed) of python and for python users.

```
a = 'Ali'
print(type(a))
```

```
<class 'str'>
```

Question

Find out 14 different `types` in python. We have already seen three types above i.e. `str`, `int` and `float`.

When we assign a value to a variable and then assign that variable to a new variable, then both of these variables actually refer to the same object. We can verify this using the function `id(VariableName)`

```
a = 12
b = a
id(a), id(b)
```

```
(139721726007952, 139721726007952)
```

Above we have checked the identity of both a and b. The identity of an object in python is the memory address of that object. It means the address in memory (RAM) where that objected is stored/put. Since both a and b refer to the same object, they have the same memory address. We can also say that since both a and b have the same memory address, that means they are same objects.

```
b = 14
id(a), id(b)
```

```
(139721726007952, 139721726008016)
```

So when we assigned a different data to b, a new object was created and now b refers to this new object and thus its identity (memory address) changes now.

Just as there are conventions for naming people in a society, there is convention and rules for naming variables in python. The name of a variable can be any alpha-numeric combination with some exceptions. Following are valid variable names

```
ali9 = 12
Ali9 = 14
 = 2
print()
```

```
2
```

A variable name must not start from a number.

```
# uncomment following line
# 1_ali = 29
```

Question: Explain the error which will result from the above code.

We can not name certain keywords as variable names. These keywords can be seen official python docs website[1]

Data Types

Data type signifies the type of operation that can be performed on that data. The three common data types in python are as follows

- numeric
- sequence
- boolean

Numbers

To represent numerical values, python has three types

- integer
- float
- complex

```
a = 1
b = 0b101   # binary with base 2
x = 0o14  # octal with base 8
y = 0xe     # hexadecimals with base 16

print(a, type(a))
print(b, type(b))
print(x, type(x))
print(y, type(y))
```

```
1 <class 'int'>
5 <class 'int'>
12 <class 'int'>
14 <class 'int'>
```

[1] https://docs.python.org/2.5/ref/keywords.html

0b101, *0o14* and *0xe* are examples of integers represented in binary, octal and hexadecimal bases respectively. We can convert a number to binary, octal and hexadecimal using the functions `bin`, `oct` and `hex` respectively as follows

```
print(bin(5))
print(oct(12))
print(hex(14))
```

```
0b101
0o14
0xe
```

However, if you don't understand the meaning of binary, octal and hexadecimal bases, don't worry. This is not important at this point. Just keep in mind that we can represent numbers in different bases in python.

```
a = 12.5e3
print(type(a))
```

```
<class 'float'>
```

Since the value of `a` is a float, the type of `a` is float. The value of `a` is 12.5 times 10 raised to the power 3. This is called scientific notation. We can also represent a number in scientific notation as follows

```
a = 12.5e-3
```

Question What is the value of `a` in the above code?

Question:

Define variables to store the information on your ID card and print each of them. The variables should not be of same type.

```
x = 3 + 4j  # consist of real and imaginary part
print(type(x))
```

```
<class 'complex'>
```

```
coke = False
water = True
print(type(coke))
```

```
<class 'bool'>
```

Question What will the suitable data type to store currency values? Explain your reasoning.

Sequence

boolean

Total running time of the script: (0 minutes 0.008 seconds)

1.2 sequential data

A sequential data is the data type with one or more than one value/object in it.

There are four major built-in sequential data types in python

- Strings
- Lists
- Tuple
- Range

Since a sequential data type consists of more than one value, we can check the length of a sequential data using `len` function.

Strings

In python, a string is a data type which does not have a numeric value and is therefore treated as a text.

```
s = 'Ali'
print(type(s))
```

```
<class 'str'>
```

The value/data of string need not to be only English language characters. They can be anythin such as numbers unless they are defined as string.

```
s = '12'
print(type(s))
```

```
<class 'str'>
```

Although, 12 above is a numeric value, but since is enclosed inside quotation marks `''`, python considers it a string and not a number.

There are four common ways to define a string in python:

- single quotation `'The wretched of the earth'`
- double quotation `"The wretched of the earth"`
- triple doubel quotations `"""The wretched of the earth"""`
- triple quotation `'''The wretched of the earth'''`

```
s1 = 'Only persons really changed history those who changed men`s thinking about␣
↪themselves'
s2 = "Only persons really changed history those who changed men's thinking about␣
↪themselves"
s3 = """Only persons really changed history those who changed men's thinking about␣
↪themselves"""
s4 = '''Only persons really changed history those who changed men's thinking about␣
↪themselves'''
print(type(s1), type(s2), type(s3), type(s4))
```

```
<class 'str'> <class 'str'> <class 'str'> <class 'str'>
```

A string can be as long as we wish!

```
s = 'What is the first question that should come to our mind in this life?'
s2 = "Should Immanuel Kant be condemned for his racist views?"

print(type(s), type(s2))
```

```
<class 'str'> <class 'str'>
```

```
s3 = 'Why the colonization isn\'t considered a crime?'
print(s3)
```

```
Why the colonization isn't considered a crime?
```

```
s3 = "Why the colonization isn't considered a crime?"
print(s3)
```

```
Why the colonization isn't considered a crime?
```

If we want to quote something with double strings inside a double quoted string, we can do it as following.

```
txt = "He said: \"It doesn't matter, if you enclose a string in single or double quotes!\
↪""
print(txt)
```

```
He said: "It doesn't matter, if you enclose a string in single or double quotes!"
```

```
txt = '''Baqir al sadr was an Iraqi scholar.
He was born in 1935 and wrote his famous book "our philosophy" just at the age of 25.
He was killed at the age of 45 by Saddam Husain.'''
print(txt)
```

```
Baqir al sadr was an Iraqi scholar.
He was born in 1935 and wrote his famous book "our philosophy" just at the age of 25.
He was killed at the age of 45 by Saddam Husain.
```

Question: Print the following sentence including the double quotations.

“The one who controls his desires is a free man. Ali ibn Abi Talib”

Indexing

Indexing refers to selecting a value at a certain position from a sequential data. It shhould be noted that in python, the indexing starts from 0 and not from 1. This means, 0 refers to 1st position and 1 refers to second position. The slice operator `[]` is used to index a sequential data. %%

```
s = "Assalam o alaikum"
print(s[0])
```

```
A
```

Above we have selected the first character of the string `s`. We can select any character by changing the index in the slice operator.

```
print(s[7])
```

We can find the length of a sequence object in python using the function *len*. For strings, we can check the number of characters `len`. The length of a string is the number of characters in it including the empty spaces if they are present. When we check the length of a string using `len` function, it returns an integer.

```
len(s)
```

```
17
```

We can also select the last character of a string using negative indexing.

```
print(s[len(s)-1], s[-1])
```

```
m m
```

Question:

Explain the difference between `s[len(s)-1]` and `s[-1]`.

Select the last character of the string `s` using positive indexing.

Slicing

We can select a sequence of characters from a string using slice `:` operator.

```
print(s[-3:], s[5:8], s[8:])
```

```
kum am  o alaikum
```

Above we are selecting and printing three three different slices of the string `s`. In the first slice, we have selected the last three characters of the string `s` using negative indexing. In the second slice, we have selected the characters from 6th to 8th position of the string `s` using positive indexing. In the third slice, we have selected the characters from 9th position to the end of the string `s` using positive indexing.

Concatenation

Concatenation in strings string refers to joining two or more strings together. This can be done using the + operator.

```
a = " Assalam" + " o" " alaikum"
print(a)
```

```
Assalam o alaikum
```

Question:

Write a code so that following two lines become a single sentence.

“The one who controls his desires”

“is a free man”

Repetition

```
b = a*3
print(b)
```

```
Assalam o alaikum Assalam o alaikum Assalam o alaikum
```

Question: Print the following string 10 times by making use of *.

“Black skin, white masks. “

immutability

```
# uncomment following line
# a[-1] = ". "  # TypeError
```

```
a = "Muhammad"
b = "Muhammad"
print(a is b)
```

```
True
```

This above line uses the `is` operator to check if the two variables `a` and `b` refer to the same object in memory. The `is` operator returns True if they do and False otherwise.

```
print(a == b)
```

```
True
```

In the above line, we are using the == operator to check if the two variables `a` and `b` have the same value. The == operator returns True if they do and False otherwise.

```
a = "Muhammad!"
b = "Muhammad!"
print(a is b)
```

```
True
```

```
print(a == b)
```

```
True
```

```
a = "Muhammad1"
b = "Muhammad1"
print(a is b)
```

```
True
```

```
print(a == b)
```

```
True
```

Lists

Lists are array likes, enclosed by square brackets.

```
a = [1, 2, 3, 4]
print(a)
```

```
[1, 2, 3, 4]
```

we can verify that variable `a` is of *list* type

```
type(a)
```

But `a` is not just an array like in *Fortran* programming language. It is rather, much more than that. It is a collection of objects.

```
type(a[0]), type(a[1])
```

```
(<class 'int'>, <class 'int'>)
```

All elements in above list were of type *int* but a list can hold any kind of objects all in same list. The following list contains, *int*, *float*, *str* and a *list* type in it. Yes a list can have a list inside it as well. That is why it is called a **collection of objects**.

```
a = [1, 2.0, '3', [4, 5]]
type(a[0]), type(a[1]), type(a[2]), type(a[3])
```

```
(<class 'int'>, <class 'float'>, <class 'str'>, <class 'list'>)
```

as we have seen above that we can index list elements as well. However, if we try to access for an index in a list which is not present, it will result in error. For example above list contains four elements. If we try to access a[4] (which means element at 4th index or 5th element), it will result in error.

```
# uncomment following line
# a[4]  # IndexError
```

We have also seen that list is an ordered collection of objects. If we print list *a*, it will print its elements in same sequence as they are originally.

```
print(a)
```

```
[1, 2.0, '3', [4, 5]]
```

We can change contents of lists using indexing operator `[]`. We have already seen that teh `[]` operator is used to create a list. It is also used to index a list and to change its contents.

```
a[3] = 'a new element'
print(a)
```

```
[1, 2.0, '3', 'a new element']
```

Above we have replaced the 4th element of list *a* with a new element. If you have not understoop this, don't worry, more of this wil come in the chapter of *1.4 lists*.

We can replace a sequence of elements in a list with a new sequence and the new sequence does not have to be of same length and type as old sequence.

```
print(a[0:3])

a[0:3] = [2.0, 2]
print(a)
```

```
[1, 2.0, '3']
[2.0, 2, 'a new element']
```

So we see the size of list is changed automatically/dynamically.

nested lists

A list can contain several lists and every list in side a list further sublists and so on.

```
pakistan = [[['Nawakali', 'Alamdar Road', 'Killi Ismail', 'Kharotabad'],
             'Kallat', 'Ziarat', 'Gawadar'],
            ['Sukkur', 'Rohri', 'Hayderabad', 'Karachi'],
            ['Peshawar', 'Hangue', 'Mardan', 'Charsadda'],
            ['Lahore', ['ugoki', 'sambrial', 'pasrur', 'daska'], 'Sadiqabad', 'Multan']]

len(pakistan)
```

```
4
```

Finding length of *pakistan* will give length of outermost list. We can find out lengths of inner lists as well.

```
len(pakistan[0]), len(pakistan[1]), len(pakistan[2]), len(pakistan[3])
```

```
(4, 4, 4, 4)
```

```
print(pakistan[0][0][0:])
```

```
['Nawakali', 'Alamdar Road', 'Killi Ismail', 'Kharotabad']
```

```
print(pakistan[3][0][:])
```

```
Lahore
```

```
print(pakistan[3][1][-3:])
```

```
['sambrial', 'pasrur', 'daska']
```

```
print(pakistan[3][1][3][3:])
```

```
ka
```

```
enigma = 'IRtu diysa rtdo oK icpolnivnegn iweanst  at ow hbiet ea  sluipbeerrmaalc!i s t'
print(enigma[::2])
```

```
It is too convenient to be a liberal!
```

Above `enigma` is a string by `enigma[::2]` we start from first value until the end with a jump of two. Note that empty space is also a valid string of length one. For example we start with `I` and then jump to the second position after `I` which is `t`. Then we again jump to the second position after `t` which is an empty space. This continues until we reach `!`. Note that there are two empty spaces when we print the output of `enigma[::2]`.

```
print(enigma[1::2])
```

```
Rudyard Kipling was a white supermacist
```

Similarly `enigma[1::2]` indicates that start from second value of `enigma` until the end with a jump of two. Here also empty space indicates a valid string of length one. So we start with the second member of `enigma` which is `R` and then jump to the second position after `R` which is `u`. This continues until we reach the last member of `enigma`. Here we started with the second member of `enigma` and not the first member because we are using `1` in the slice operator.

concatenation

List concatenation works same as that of strings.

```
provinces = ['sind', 'balochistan', 'kpk', 'punjab'] + ['janubi punjab', 'kashmir',
↪'potohar']
print(provinces)
```

```
['sind', 'balochistan', 'kpk', 'punjab', 'janubi punjab', 'kashmir', 'potohar']
```

```
provinces += ['hazara', 'karachi']
print(provinces)
```

```
['sind', 'balochistan', 'kpk', 'punjab', 'janubi punjab', 'kashmir', 'potohar', 'hazara',
↪ 'karachi']
```

Tuples

In contrast to lists, tuples are immutables which means, once they are created, we can not change/modify their content.

```
panjtan = (1,2,3,4,5)
type(panjtan)
```

Just like lists, the contents of tuples can also be of different types.

```
panjtan = (1, 2, 'three', 4.0, [5])
print(panjtan)
```

```
(1, 2, 'three', 4.0, [5])
```

We can not change a value in a tuple.

```
# uncomment following line
# panjtan[2] = 'Musa'  # TypeError
```

Question:

Explain what were trying to acheive by `panjtan[2] = 'Musa'` and why it resulted in error. On the otherhand, if we do `panjtan[2]`, why it will not result in error?

Tuples are used to store data where we know it will not change. This also makes sure that we don't change the data accidentally.

in

The *in* is a buil-tin keyword which is used to check whether an element is present in a sequence or not.

```
print("Bahawalpur" in provinces)
```

```
False
```

We had created a list *provinces* above. We are checking if "Bahawalpur" is present in it or not in the above code. The output from above code will be *False* because "Bahawalpur" is not present in the list *provinces*.

```
print("Multan" not in provinces)
```

```
True
```

```
print("pubjab" in provinces)
```

```
False
```

We are checking if "pubjab" is present in the list *provinces* or not.

We can also use *in* to check if a string is present in a tuple or not.

```
print('three' in panjtan)
```

```
True
```

Similarly we can use *in* to check if a string is present in a string or not.

```
print('at' in enigma)
```

```
True
```

Question: Consider the following python list.

```
scoundrels = ['asim', 'kakar', 'mohsin']
```

Write code to using *in* to check if *bajwa* is a scoundrel or not?.

Repetition

```
text = ["Gaza is an open air prison. "]
t = [text] * 3
print(t)
```

```
[['Gaza is an open air prison. '], ['Gaza is an open air prison. '], ['Gaza is an open␣
↪air prison. ']]
```

Caveat

```
print(t[0][0])
```

```
Gaza is an open air prison.
```

```
t[0][0] = "Yemen is an open air prison. "
print(t)
```

```
[['Yemen is an open air prison. '], ['Yemen is an open air prison. '], ['Yemen is an␣
↪open air prison. ']]
```

t[0][0] points to first member of the first member of *t*. So when we did *t[0][0]=something*, we were assigning new value at that position.

Therefore, the original contents of *t* also changed.

However, since all the three members of *t* are pointing to the same memory location. So if we change any of the members, all will change. This is called **shallow copy**. This is because because *t[1]* is same as *t[0]* and not a *deep copy* of *t[0]*. *t[1]* and *t[0]* are indeed same objects and points to same position in memory. We can say that *t[1]*, *t[0]* and *[t2]* are all `text`. This is because of the way we created t. By [text]*3 did not (make a deep) copy (of) the conents of the list *text*. We will study more about this in the chapter of *1.5 copying lists*.

Indexing

```
# a = ['Makran',' coastal', 'highway', 'in', 'Balochistan', 'is', 'stunning']
a = "Lasbela and Loralai!"
# a = ('Makran',' coastal', 'highway', 'in', 'Balochistan', 'is', 'stunning')

start = 2
stop = 7
print(a[start:stop])  # items start through stop-1
print(a[start:])      # items start through the rest of the array
print(a[:stop])       # items from the beginning through stop-1
print(a[:])           # every item in sequence
```

```
sbela
sbela and Loralai!
Lasbela
Lasbela and Loralai!
```

```
print(a[-1])    # last item in the sequence
print(a[-2:])   # last two items in the sequence
print(a[:-2])   # whole sequence
```

```
!
i!
Lasbela and Lorala
```

```
print(a[::-1])    # all items in the sequence, reversed
print(a[-3::-1])  # starting with the 3rd item from the end, all items in the sequence,␣
↪reversed
```

```
!ialaroL dna alebsaL
alaroL dna alebsaL
```

Range

It gives immutable sequence. We will further study its use later during in *1.10 for loops*.

```
a = range(4)
print(a)
```

```
range(0, 4)
```

```
print(type(a))
```

```
<class 'range'>
```

Total running time of the script: (0 minutes 0.019 seconds)

1.3 operators

This lesson describes Basic operators in python.

The traiditional mathematical operations can be performed on python objects just by using their symbels. This means addition, subtraction, multiplication and divitsion can be perfomred just by using +, -, * and / symbols/operators between two objects respectively.

Basic operations

+ is used for addition

```
a = 5 + 12
print(a)
```

```
17
```

If we want to add something to variable 'a', one way of doint it is using +=.

```
a +=14
print(a)
```

```
31
```

Above we added 10 to *a* and assigned the new value again to *a*. In this way, the value of *a* is updated.

Similarly we can use - for subtracting one object from another

```
a = 14 - 12
print(a)
```

```
2
```

```
a -= 5
print(a)
```

```
-3
```

* is used for for multiplication

```
a = 2*12
print(a)
```

```
24
```

```
a *= 2
print(a)
```

```
48
```

The modulo operator % returns remainder

```
print(14 % 5)
```

```
4
```

If one of the value is float, result will be a float.

```
print(17 % 5.0)
```

```
2.0
```

Above *17* was integer and *5.0* was float. Since one value was float, the result is also a float

The sign of the result will be same as sign of divider.

```
print(17 % -5.0)
```

```
-3.0
```

/ is used for division

```
a = 20/6
print(a)
```

```
3.3333333333333335
```

// is used for truncated division

```
print(20//6)
```

```
3
```

```
print(20//6.0)
```

```
3.0
```

If the answer of the truncated division is negative, the answer is rounded to the next smallest iteger (greater negative)

```
print(20//-6.0, -20//6.0)
```

```
-4.0 -4.0
```

```
print(-20 // -6.0)
```

```
3.0
```

Question What is the output of the following code

```
print(20//-6.0, -20//6.0)
```

Question What is the output of the following code

```
print( 20//60, 20/60)
```

** is used for exponentiation i.e. to raise one object over another. For example two to the power of three can be done as below

```
print(2**3)
```

```
8
```

The application of basic mathematical operators is not limited to floats or integers. We can also apply + to strings in python

```
a = "Materialism leads to "
b = "injustice."
print(a + b)
```

```
Materialism leads to injustice.
```

However we can not do same for - operator. This means we can not subtract two strings.

```
# uncomment following line
# a - b # -> TypeError
```

But when we multiply a `string` by an `integer`, the string is repeated.

```
print(a * 2)
```

```
Materialism leads to Materialism leads to
```

Above we are multiplying a string with an integer because *a* is a string and *2* is integer. If however, we do `a +2`, we will again get `TypeError`.

What kind of mathematical operations can be applied on an object or between two objects, depends purely upon the objects. To our surprise, we can even modify the behavior of these mathematical operators in python. More about this will come later in *3.16 magic methods*.

Comparisons

If we want to compare one object with another and tell whether both are equal or not, we make use of == operator.

```
print(2 == 3)
```

```
False
```

The == operator returns either True or False depending upon the values being compared.

```
print(2 ==2)
```

```
True
```

```
print(2.2 == 1.1 + 1.1)
```

```
True
```

However, we should avoid comparing floats in this way. This is because computers can not represent accurate values of floats. 1.1 + 2.2 results in an approximated answer, so we better avoid comparing floats.

Question Find out the type of the output returned by == operator. type(2==3)

```
print(3.3 == 1.1 + 2.2)
```

```
False
```

```
print(abs((1.1 + 2.2) - 3.3) < 1e-15)
```

```
True
```

> can be read as *greater than* or *smaller than* ` depending upon its direction.

%% If you are getting the output as `True` for the above code, it is because of the subtle differences in floating point implementation in your hardware.

If we want to check whether a numerical value lies between two numbers or not we can make use of < or > operators twice.

```
print(8<10<12)
```

```
True
```

!= can be considered as opposite of == and can be read as *not equal to*

```
capitalism = 'a system based on individualism'

print(capitalism != 'justice')
```

```
True
```

Logical operators

`not` results in opposite to what comes after it.

```
print(not True)
```

```
False
```

```
print(not False)
```

```
True
```

```
x = 5.2
print(not x<=10)
```

```
False
```

```
capitalism = False
communism = False
justice = True
print(capitalism and communism)
```

```
False
```

```
print(capitalism or communism)
```

```
False
```

```
print(capitalism and justice)
```

```
False
```

Above, since one statement on left side of `and` is False, the output is False. For `and` to return True, both statements/expressions on its right and left side, must return True. If one is True and one is False, the output will be False. However, for `or`, even if one side evaluates to True, the output is True. This becomes clear if you understand, why the output of below code is True.

```
print(capitalism or justice)
```

```
True
```

```
print(capitalism is not justice)
```

```
True
```

Question The above code `print(capitalism is not justice)` printed True. This is because `justice` was True and thus `not justice` becomes False. Therefore, what we are implicitly saying *False is False*, which is True. Now reset the values of variables *capitalism* and *justice* in such a way that the code `print(capitalism is not justice)` should return False. This is shown below

```
capitalism =
justice =
print(capitalism is not justice)  # must print False
```

Default values

```
food = 'bread'
lunch = food or 'curry'
print(lunch)
```

```
bread
```

The statement on left of `or` was not False/None, therefore, variable *lunch* became *bread* .

```
food = None
lunch = food or 'curry'
print(lunch)
```

```
curry
```

If the first argument before `or` is `True` or not None, the value after `or` is discarded.

```
food = None
lunch = 'currey' or food
print(lunch)
```

```
currey
```

```
food = 'bread'
lunch = 'currey' or food
print(lunch)
```

```
currey
```

Identity

`is` operator compares whether both variables on its right and left side refer to same memory location or not.

```
a = 257
b = 257
print(a == b)
```

```
True
```

```
id(a), id(b)
```

```
(139721348137264, 139721348137264)
```

Because ali and hasan are stored at different location at different location, thus the answer is False

```
print(a is b)
```

```
True
```

Note If the above code prints `True`, it could be that the python version that you are using saves even large integers in memory.

Actually, python already stores some commonly used smaller numbers in memory, so when they are created, python refers to that same memory location and does not really create them. Thus, for smaller numbers (from -5 to 256 integers) *is* returns True.

```
a = 256
b = 254 + 2
print(a is b)
```

```
True
```

```
print(id(a), id(b))
```

```
139721726204304 139721726204304
```

```
feudalism = 'slavery'
capitalism = 'slavery'

print(feudalism is capitalism)
```

```
True
```

```
feudalism = 'a system of slavery'
capitalism = 'a system of slavery'

print(feudalism is capitalism)
```

```
True
```

Order of operations

The multiplication * is performed before addition + even if + appears before *.

```
print(20 + 4 * 10)
```

```
60
```

Similarly *exponentiation* (**) is performed before multiplication.

```
print(2 * 3 ** 4 * 5)
```

```
810
```

Question: Write the code to convert 1000 seconds into hours.

Question: Why the output of `2/3+4` and `2/(3+4)` is different?

Complete order of precedence of operators in python can be found from here .

Total running time of the script: (0 minutes 0.014 seconds)

1.4 lists

This lesson describes the concept of *list* in python.

A list can be defined as a collection of objects or a container in which we hold different objects.

```
mylist = []
```

Above we defined an empty list. How do we know that it is list? We can always check the type of object in python as below.

```
print(type(mylist))
```

```
<class 'list'>
```

We can also make a list which is not empty as below

```
imperialists = ['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola']
print(type(imperialists))
```

```
<class 'list'>
```

In above-mentioned list, all of its (6) elements are strings. However, the elements/members a list are not required to be of same *type*.

```
imperialists = ["Bush", {"years": 8}, 2000, (2000, 2008)]
print(type(imperialists))
```

```
<class 'list'>
```

In above list, the first member is `string`, the second member is `dictionary`, the third member is `integer` and the fourth member is a `tuple`. We will study about string, dictionary, integer and tuple in upcoming lessons.

There are two ways to convert a python object into list

- using `[]`
- usnig `list` function

```
a = 1,2
print(type(a))
a_as_list_using_slice_op = [a]
print(type(a_as_list_using_slice_op))
```

```
<class 'tuple'>
<class 'list'>
```

```
a_as_list_using_list_fn = list(a)
print(type(a_as_list_using_list_fn))
```

```
<class 'list'>
```

However there is a major difference in these two. `[]` converts the whole object *as it is* into a list. On the other hand, `list` function is more like *element wise* operator. This can be verified by printing the converted lists created above.

```
print(a_as_list_using_slice_op)
print(a_as_list_using_list_fn)
```

```
[(1, 2)]
[1, 2]
```

a_as_list_using_list_fn is a list with two members while *a_as_list_using_slice_op* is a list with only one member. This can be verified by checking the length of both lists.

```
print(len(a_as_list_using_slice_op))
print(len(a_as_list_using_list_fn))
```

```
1
2
```

Then length of list which is created using slice operator `[]` is always 1, while the length of list created using `list` function depends upon the number of values in the object. The slice operator is creating a list with one member and that one member is a tuple in this case.

Spend some minutes on understanding the difference between these two ways of creating a list. Try with different objects and see the difference.

Question Write code to prove the above statement.

slicing a list

Since the list is a sequence, we can slice it as well. The slicing of a list can be done using the slice operator `[]`. Consider the following list

```
imperialists = ['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola']
print(imperialists[0])
```

```
Bush
```

```
print(imperialists[2])
```

```
Trump
```

```
print(imperialists[-1])
```

```
coca cola
```

The `:` operator/symbol is used to slice a list. The syntax is as follows `list[start:stop:step]`. The default value of `start` is 0, of `stop` is length of list and of `step` is 1.

```
print(imperialists[2:4])
```

```
['Trump', 'Zuckerberg']
```

Question: What will be the output of the following code

```
a = [1,5, 7,14]
a[2] = 12
print(a)
```

Operations on a list

Once we have a list, we can perform different operations on it. Some of them are given below.

append

```
imperialists = ['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola']
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola']
```

```
imperialists.append('clinton')
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton']
```

append changes the original list and it itself returns *None*.

```
new_imperialists = imperialists.append('netanyahu')
print(new_imperialists)
```

```
None
```

```
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', 'netanyahu']
```

If we add a similar but new element in list, then the list will have 2 such elements as its member.

```
imperialists.append('netanyahu')
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', 'netanyahu',
↪'netanyahu']
```

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
a.append(12)
print(a[-1])
```

pop

```
last_element = imperialists.pop(-1)
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', 'netanyahu']
```

```
print(last_element)
```

```
netanyahu
```

```
# uncomment following 1 line
# imperialists.pop(8)
```

```
imperialists.pop()
```

```
'netanyahu'
```

```
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton']
```

```
# uncomment following 1 line
# imperialists.pop('Bush')
```

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
a.pop(2)
print(len(a))  # ??
a.pop(2)   # ??
```

extend

If we want to add multiple elements to a list, using *append* will put a new list in the previous list

```
imperialists = ['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton']
uk_imperialists = ['churchil', 'Tony Blair', 'BBC']
imperialists.append(uk_imperialists)
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', ['churchil',
↪'Tony Blair', 'BBC']]
```

```
imperialists = ['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton']
imperialists.extend(uk_imperialists)
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', 'churchil',
↪'Tony Blair', 'BBC']
```

`extend` actually takes any sequence as input. It must not be a list. It can be a string or tuple.

```
new_one = "Murdoch"
imperialists.extend(new_one)
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'Zuckerberg', 'Bezos', 'coca cola', 'clinton', 'churchil',
↪'Tony Blair', 'BBC', 'M', 'u', 'r', 'd', 'o', 'c', 'h']
```

```
imperialists = ['Bush', 'Obama', 'Trump', 'clinton']
capitalists = ('Zuckerberg', 'Bezos', 'coca cola')
imperialists.extend(capitalists)
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola']
```

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
b = [2, 3]
print(len(a.extend(b)))  # ?
```

using + operator

We can also append lists using the + operator.

```
media_houses = ['bbc', 'cnn', 'reuters', 'springer']
print(imperialists + media_houses)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola', 'bbc', 'cnn',
↪'reuters', 'springer']
```

So when we use + operator between two lists, a new list is created which contains all the members of both lists. The original lists remain unchanged.

```
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola']
```

```
imperialists = imperialists + media_houses
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola', 'bbc', 'cnn',
↪'reuters', 'springer']
```

Above we are recreating the list *imperialists* by adding *media_houses* to it.

```
morons = ['sam haris', 'richard dawkins', 'baghdadi', 'bin ladan']
imperialists += morons
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola', 'bbc', 'cnn',
↪'reuters', 'springer', 'sam haris', 'richard dawkins', 'baghdadi', 'bin ladan']
```

+= operator means that the list on the left side of += operator is updated by adding the list on the right side of += operator.

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
b = [2, 3]
print(len(a + b))  # ?
```

remove

```
imperialists.remove('springer')
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola', 'bbc', 'cnn',
↪'reuters', 'sam haris', 'richard dawkins', 'baghdadi', 'bin ladan']
```

If we repeat the above operation, it will result in error because *springer* has already been removed from the list *imperialists*.

```
# uncomment following 2 line
# imperialists.remove('springer')
# print(imperialists)
```

insert

puts an the value before the index

```
imperialists.insert(-1, 'DW')
print(imperialists)
```

```
['Bush', 'Obama', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola', 'bbc', 'cnn',
↪'reuters', 'sam haris', 'richard dawkins', 'baghdadi', 'DW', 'bin ladan']
```

Finding position/index of a member of a list

```
imperialists.index('bbc')
```

```
7
```

```
# uncomment following 1 line
# imperialists.index('bbc', 8)
```

```
# uncomment following 1 line
# imperialists.index('bbc', 3, 6)
```

if an element is present in a list twice, index of its first position is returned.

```
last_value = imperialists[-1]
imperialists.insert(2, last_value)
print(imperialists)
```

```
['Bush', 'Obama', 'bin ladan', 'Trump', 'clinton', 'Zuckerberg', 'Bezos', 'coca cola',
↪'bbc', 'cnn', 'reuters', 'sam haris', 'richard dawkins', 'baghdadi', 'DW', 'bin ladan']
```

```
imperialists.index(last_value)
```

```
2
```

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
a.insert(2, 12)
print(a)  # ?
```

reverse

```
imperialists = ['bbc', 'cnn', 'reuters', 'springer', 'voa']
imperialists.reverse()
```

The function does not return anything itself but the original list is reversed.

```
print(imperialists)
```

```
['voa', 'springer', 'reuters', 'cnn', 'bbc']
```

Question: What will be the output of the following code?

```
a = [1,5, 7,14]
a.reverse()
print(a)  # ?
```

sort We can sort the list using the *sort* function. If the contents of the list strings, then the list will be sorted in lexicographical order. If the contents are numbers, then the list will be sorted in ascending order.

```
print(imperialists)
```

```
['voa', 'springer', 'reuters', 'cnn', 'bbc']
```

The function does not return anything itself but the original list is sorted.

```
imperialists.sort()
```

```
print(imperialists)
```

```
['bbc', 'cnn', 'reuters', 'springer', 'voa']
```

```
imperialists.sort(reverse=True)
print(imperialists)
```

```
['voa', 'springer', 'reuters', 'cnn', 'bbc']
```

```
numbers = [4,3,6,2]
numbers.sort()
print(numbers)
```

```
[2, 3, 4, 6]
```

```
# uncomment following 2 line
# imperialists = ['bbc', 1, 'cnn', 3, 'voa', 2]
# imperialists.sort()
```

Question What will be the output of the following code

```
x = [1,2]
y = [3,4, 5]
print(len(x + y))
```

*

```
a = ['Najaf']
print(a * 3)
```

```
['Najaf', 'Najaf', 'Najaf']
```

```
a = ['Najaf', '->', 'Karbala']
print(a * 3)
```

```
['Najaf', '->', 'Karbala', 'Najaf', '->', 'Karbala', 'Najaf', '->', 'Karbala']
```

Notes

I have been using the word function for `append`, `sort` etc. However, in Object-Oriented Programming, it can be seen that they are actually called *methods*.

There are a lot more powerful list manipulations which can be done by combining conditional and looping statements. We will come back to them once looping and conditioning statements are covered.

Total running time of the script: (0 minutes 0.014 seconds)

1.5 copying lists

This lessor describes how to copy a list in Python.

Important: This lesson is still under development.

Suppose we have a list of countries

```
countries1 = ["Iraq", "Iran", "Lebanon"]
countries2 = countries1

print(countries1)

print(countries2)
```

```
['Iraq', 'Iran', 'Lebanon']
['Iraq', 'Iran', 'Lebanon']
```

Above we have created a new list *countries2* and assigned the list *countries1* to it. Now both the lists *countries1* and *countries2* are same. We can check this by checking their memory address.

```
print(id(countries1),id(countries2))
```

```
139721349008256 139721349008256
```

We can see, both the lists *countries1* and *countries2* point to same object in memory. This means, even though we have two list variables, in reality we have only one object.

What will happen if we redfine the list *countries2*?

```
countries2 = ["Qatar", "Malaysia", "Turkey"]
print(countries1)

print(countries2)
```

```
['Iraq', 'Iran', 'Lebanon']
['Qatar', 'Malaysia', 'Turkey']
```

```
print(id(countries1),id(countries2))
```

```
139721349008256 139721348554496
```

Now *countries1* and *countries2* are different objects. So the *countries2* list became different from *countries1* when we assigned a different object to it. What happens when we change the contents of list

```
countries1 = ["Pakistan", "Iran", "Turkey"]
countries2 = countries1

countries2[2] = "Syria"

print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Syria']
['Pakistan', 'Iran', 'Syria']
```

Even though we only changed *countries2*, the list *countries1* changed automatically. This is because, we did't have two lists to begin with. We have one list with two names. What we did was *in place* change in *countries2* list and did not assign a new object to *countries2*, so the the name *countries2* still points to the same object as *countries1*.

Question: What will be the output of the following code?

```
nasalkush_countries = ["us", "uk", "germany"]
istemari_countries = nasalkush_countries

nasalkush_countries.extend(["france"])

print(istemari_countries)
```

Copying using slicing

The above problem can be avoided by using the slice operator

```
countries1 = ["Pakistan", "Iran", "Turkey"]
countries2 = countries1[:]
countries2[2] = 'Syria'

print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey']
['Pakistan', 'Iran', 'Syria']
```

```
print(id(countries1), id(countries2))
```

```
139721348995776 139721349003520
```

Now *countries1* and *countries2* are not same objects. %% md

> The same can be achieved by using *copy* method of *list* object.

```
countries1 = ["Pakistan", "Iran", "Turkey"]
countries2 = countries1.copy()

print(id(countries1), id(countries2))

# What about sublists in a list?
```

```
139721349010176 139721348995776
```

```
countries1 = ["Pakistan", "Iran", "Turkey",["Qatar", "Malaysia"]]
countries2 = countries1[:]

print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Malaysia']]
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Malaysia']]
```

```
countries2[0] = "Syria"
print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Malaysia']]
['Syria', 'Iran', 'Turkey', ['Qatar', 'Malaysia']]
```

```
countries2[3][1] = "Iraq"
print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Iraq']]
['Syria', 'Iran', 'Turkey', ['Qatar', 'Iraq']]
```

So again *countries1* is changed even though we changed only *countries2* list. This is because when we copy a list containing sublists, the sublists are not copied but their reference is copied.

We can check that the sublists in both lists (countries1 and countries1) are same objects.

```
print(id(countries1[3]), id(countries2[3]))
```

```
139721349010688 139721349010688
```

The same can problem arisis when use *copy* method of *list* object.

```
countries1 = ["Pakistan", "Iran", "Turkey",["Qatar", "Malaysia"]]
countries2 = countries1.copy()

print(id(countries1), id(countries2))
```

```
139721349009600 139721349010752
```

```
countries2[3][1] = "Iraq"

print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Iraq']]
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Iraq']]
```

```
print(id(countries1[3]), id(countries2[3]))
```

```
139721349011392 139721349011392
```

Using `list` function

The `list` function converts a sequence into list, if it is not already a list

```
countries1 = ["Pakistan", "Iran", "Turkey",["Qatar", "Malaysia"]]
countries2 = list(countries1)

print(id(countries1), id(countries2))
```

```
139721348995776 139721348824576
```

```
print(id(countries1[3]), id(countries2[3]))
```

```
139721349012096 139721349012096
```

using copy module

```
from copy import copy

countries1 = ["Pakistan", "Iran", "Turkey",["Qatar", "Malaysia"]]
countries2 = copy(countries1)

print(id(countries1), id(countries2))
```

```
139721348554432 139721348995776
```

```
print(id(countries1[3]), id(countries2[3]))
```

```
139721348526400 139721348526400
```

```
from copy import deepcopy

countries1 = ["Pakistan", "Iran", "Turkey",["Qatar", "Malaysia"]]

countries2 = deepcopy(countries1)

print(id(countries1), id(countries2))
```

```
139721349012288 139721348554432
```

Now *countries1* and *countries2* are different and also the sublists in both lists are different now.

```
print(id(countries1[3]), id(countries2[3]))
```

```
139721350848704 139721352060480
```

So if we make change in one sublist, the sublist in other list will not be affected.

```
countries2[3][1] = "Iraq"

print(countries1)

print(countries2)
```

```
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Malaysia']]
['Pakistan', 'Iran', 'Turkey', ['Qatar', 'Iraq']]
```

but the strings in them are not copied so they still have same id.

```
print(id(countries1[0]), id(countries2[0]))
```

```
139721349009840 139721349009840
```

So `deepcopy` method from *copy* module is the safest way to copy a list when it contain sublists but it is also the slowest one among others. But what if the two sublists in a list are themselves same objects?

```
countries1 = ["Pakistan", "Iran"]
b = [countries1,countries1] # there's only 1 object countries1
c = deepcopy(b)

# check the result
c[0] is countries1 # return False, a new object a' is created
```

```
False
```

```
c[0] is c[1]
```

```
True
```

In such a scenario, copying with nested for loops is the safest way. *for loops* will be described later.

```
countries1 = ["Pakistan", "Iran"]
b = [countries1,countries1] # there's only 1 object countries1

c = [[i for i in row] for row in b]
```

```
c[0] is countries1
```

```
False
```

```
c[0] is c[1]
```

```
False
```

If you don't understand the above code, don't worry. We will discuss it later. For now, just remember that if you have a list containing lists, then *deepcopy* method will not work as expected.

Total running time of the script: (0 minutes 0.010 seconds)

1.6 dictionary

This lesson describes a special data structure of python called `dictionary`.

Intro

Dictionaries are data containers that store the data as key, value pairs. Each value in a dictionary is associated with a key, and threfore every key must have a value associated with it. Therefore, dictionaries are also sometimes known as associative arrays.

We can define a dictionary using curly brackets "{}". Each key and value pair must be separated by a comma "," while a colon ":" is used to separate a key from its vlaue.

```
man = {"name": "Baqir -al- Sadr",
       "born": 1935,
       "citizenship": "Iraq",
       "died": 1979,
       "0": 0
       }
```

We can verify that whether a python object is dictionary or not by checking its `type`. A variable which is a dictionary has a `dict` type.

```
print(type(man))
```

```
<class 'dict'>
```

We can access data from a dictionary by making use of slice operator `[]`.

```
print(man["name"])
```

```
Baqir -al- Sadr
```

Inside the square bracket, we can write any `key` which is present in the dicionary and we will get the value associated with it.

```
print(man["citizenship"])
```

```
Iraq
```

If we try to a access value whose corresponding key does not exist in the dictionary, we will get `KeyError`.

```
# uncomment following line
# man["city"]  # -> KeyError
```

The error suggests that the dictionary `man` does not have a key named *city*.

The key must be the same object as when it was defined. We can not use indexing for keys such as

```
# uncomment following line
# man[0]  # KeyError
```

The `man` key does have a key with the name '0' but this is string type and we provide 0 as integer and threfore we got KeyError.

We can add a new key, value pair in an existing dictionary as following

```
man["city"] = "Baghdad"

print(man)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1979, '0': 0,
↪'city': 'Baghdad'}
```

Thus we can start with an empty dictionary and populate it later on

```
man = {}

print(man)
```

```
{}
```

```
man["city"] = "Baghdad"
man["name"] = "Baqir -al- Sadr"
man["born"] = 1935
man["citizenship"] = "Iraq"
man["died"] = 1979

print(man)
```

```
{'city': 'Baghdad', 'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died
↪': 1979}
```

The values to different keys in a dictionary can be same.

```
man["birth_place"] = "Baghdad"
man["death_place"] = "Baghdad"
```

(continues on next page)

(continued from previous page)

```
print(man)
```

```
{'city': 'Baghdad', 'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died
↪': 1979, 'birth_place': 'Baghdad', 'death_place': 'Baghdad'}
```

But we can not have a dictionary with two or more same keys. If we add a new key with same name, the previous key, value will be replaced.

```
man["died"] = 1980
print(man)
```

```
{'city': 'Baghdad', 'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died
↪': 1980, 'birth_place': 'Baghdad', 'death_place': 'Baghdad'}
```

type of keys and values

The values in a dictionary can be of any type.

```
colonies = {"british": ["India", "Australia"],
            "french": "Libya",
            "polish": 0,
            "german": 3.5,  # most of their colonies are split and joined into new␣
↪countires.
            "cuba": None}

print(colonies)
```

```
{'british': ['India', 'Australia'], 'french': 'Libya', 'polish': 0, 'german': 3.5, 'cuba
↪': None}
```

But the keys of a dictionary must be immutable.

```
persons = {1: "Adam",
           "Two": "Eva"}

print(persons)
```

```
{1: 'Adam', 'Two': 'Eva'}
```

```
# uncomment following line
# persons[[0, 1]] = ["Adam", "Eva"]  # TypeError
```

Making a real practical dictionary

```
ur_per = {"admi": "mard", "aurat": "zan", "bacha": "tefl", "paighambar": "paighambar"}
per_ar = {"mard": "rojol", "zan": "nissa", "tefl": "tefl", "paighambar": "paighambar"}

print("The Arabic word for aurat is: " + per_ar[ur_per["aurat"]])
```

```
The Arabic word for aurat is: nissa
```

keys and values methods

We can get all the keys of a dictionary using `.keys()` method on dictionary. The dot "." here signifies that the `keys()` function comes from dictionary. This means any variable which is a dictionary, will have .keys() function in it.

```
books = {"AlSadr": ["Our Philosophy", "Our Economy"],
         "Mutahri": ["Divine Justice", "Man and Destiny"]
         }
keys = books.keys()

print(keys)
```

```
dict_keys(['AlSadr', 'Mutahri'])
```

Algthouh the printing keys look like `list` but in reality their time is not `list`.

```
print(type(keys))
```

```
<class 'dict_keys'>
```

We can convert keys of a dictionary into `list` type as follows

```
keys_as_list = list(keys)
print(type(keys_as_list))
```

```
<class 'list'>
```

```
print(keys_as_list)
```

```
['AlSadr', 'Mutahri']
```

Similarly we can convert values of a dictionary into `list` type as follows

```
values = books.values()

values = list(values)

print(values)

print(type(values))
```

```
[['Our Philosophy', 'Our Economy'], ['Divine Justice', 'Man and Destiny']]
<class 'list'>
```

Question: Consider the following two dictionries:

```
a = {"Ali": 661, "Hassan": 670, "Hussain": 680, "Abid": 712, "Baqir": 733, "Jafar": 765}

b = {"Musa": 799, "Raza": 818, "Taqi": 835, "Naqi": 868, "Hassan": 874, "Hussain": None}
```

Write the code which will return a single list of values from both dictionaries i.e. the code should return `[661, 670, 680, 712, 733, 765, 799, 818, 835, 868, 874, None]`.

The function `items()` when applied on a dictionay, returns both keys and values.

```
items = books.items()
print(items)
```

```
dict_items([('AlSadr', ['Our Philosophy', 'Our Economy']), ('Mutahri', ['Divine Justice',
↪ 'Man and Destiny'])])
```

```
print(type(items))
```

```
<class 'dict_items'>
```

```
books_as_list = list(books.items())

print(books_as_list)
```

```
[('AlSadr', ['Our Philosophy', 'Our Economy']), ('Mutahri', ['Divine Justice', 'Man and␣
↪Destiny'])]
```

Each member of `books_as_list` is a tuple of two elements (keys and values).

```
print(type(books_as_list[0]))
```

```
<class 'tuple'>
```

```
print(books_as_list[0])
```

```
('AlSadr', ['Our Philosophy', 'Our Economy'])
```

dictionaries from lists

If we have two lists and we want to convert them into a dictionary, in such a way that the first list becomes keys and the second list becomes values, we can do this as follows

```
provinces_capitals_dict = dict(
    list(zip(["Balochistan", "Sindh", "KPK", "Punjab"], ["Quetta", "Karachi", "Peshawar",
↪ "Lahore"])))

print(provinces_capitals_dict)
```

```
{'Balochistan': 'Quetta', 'Sindh': 'Karachi', 'KPK': 'Peshawar', 'Punjab': 'Lahore'}
```

We can also use the following code to achieve the same result

```
capitals = ["Quetta", "Karachi", "Peshawar", "Lahore"]
provinces = ["Balochistan", "Sindh", "KPK", "Punjab"]

dict(zip(provinces, capitals))
```

```
{'Balochistan': 'Quetta', 'Sindh': 'Karachi', 'KPK': 'Peshawar', 'Punjab': 'Lahore'}
```

In above code, we are making use of `zip`, `list` and `dict` functions together. This working cab be broken down into following steps:

First we make use of `zip` function to convert these two lists into a generator. More about generators and zip will come later.

```
provinces_capitals_iterator = zip(provinces, capitals)

print(provinces_capitals_iterator)
```

```
<zip object at 0x7f1369508740>
```

```
provinces_capitals = list(provinces_capitals_iterator)
print(provinces_capitals)
```

```
[('Balochistan', 'Quetta'), ('Sindh', 'Karachi'), ('KPK', 'Peshawar'), ('Punjab', 'Lahore
↪')]
```

Now we can convert *provinces_capitals* into dictionary by making use of `dict` function.

```
provinces_capitals_dict = dict(provinces_capitals)
print(provinces_capitals_dict)
```

```
{'Balochistan': 'Quetta', 'Sindh': 'Karachi', 'KPK': 'Peshawar', 'Punjab': 'Lahore'}
```

Operations on dictionaries

Following examples show, how to apply different operations with dictionaries.

len

```
man = {"name": "Baqir -al- Sadr",
       "born": 1935,
       "citizenship": "Iraq",
       "died": 1979}

len(man)
```

```
4
```

in

```
print("died" in man)
```

```
True
```

```
del man["died"]
print("died" not in man)
```

```
True
```

Repeating the above code will result in error.

We can also combine `in` with `not`

```
print("city" not in man)
```

```
True
```

pop

```
man = {"name": "Baqir -al- Sadr",
       "born": 1935,
       "citizenship": "Iraq",
       "died": 1979}

man.pop("died")
```

```
1979
```

```
print(man)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq'}
```

If we try to remove a non-existing key using pop, it will throw KeyError.

```
# uncomment following line
# man.pop("died")  # KeyError
```

However, we can avoid this error by supplying the default value that needs to be returned.

```
man.pop("dob", 19350101)
```

```
19350101
```

Therefore, we can use this method to avoid the error of removing the key from a dictionary if the key is not present in dictionary.

```
man.pop("died", None)
```

```
print(man)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq'}
```

poopitem

```
man = {"name": "Baqir -al- Sadr",
       "born": 1935,
       "citizenship": "Iraq",
       "died": 1979}

man.popitem()
```

```
('died', 1979)
```

```
print(man)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq'}
```

```
man.popitem()
```

```
('citizenship', 'Iraq')
```

```
print(man)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935}
```

get

This method can also be used for accessing the values in dictionary. It returns *None* if the key is not present and we can set the default value for a key if the value is not already present.

```
man = {"name": "Baqir -al- Sadr",
       "born": 1935,
       "citizenship": "Iraq",
       "died": 1979}

print(man.get("city"))
```

```
None
```

```
man.get("city", "Baghdad")
```

```
'Baghdad'
```

If a key is not present in a dictionary, and we try to access its value, we can avoid the KeyError by setting the default value of that key

```
print(man.get('age', 44))
```

```
44
```

copy

Simple object assignment with = makes a shallow copy.

```
man1 = {"name": "Baqir -al- Sadr",
        "born": 1935,
        "citizenship": "Iraq",
        "died": 1979}

man2 = man1
man2["name"] = "Mutahri"

print(man1)
```

```
{'name': 'Mutahri', 'born': 1935, 'citizenship': 'Iraq', 'died': 1979}
```

```
print(man2)
```

```
{'name': 'Mutahri', 'born': 1935, 'citizenship': 'Iraq', 'died': 1979}
```

so even though we changed the `name` of `man2`, but `name` of `man1` is also changed.

```
man1 = {"name": "Baqir -al- Sadr",
        "born": 1935,
        "citizenship": "Iraq",
        "died": 1979}

man2 = man1.copy()

man2["name"] = "Mutahri"
```

```
print(man1)
print(man2)
```

```
{'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1979}
{'name': 'Mutahri', 'born': 1935, 'citizenship': 'Iraq', 'died': 1979}
```

Now we don't see `name` of `man1` dictionary from getting changed. This is because we made a copy of `man1` and set this copy to `man2`. After that we changed `man2` key.

```
men1 = {1: {"name": "Baqir -al- Sadr", "born": 1935, "citizenship": "Iraq", "died": 1980}
↪,
        2: {"name": "Mutahri", "born": 1919, "citizenship": "Iran", "died": 1979}}

men2 = men1.copy()
```

(continues on next page)

(continued from previous page)

```
men2[2]["name"] = "Murtaza Mutahri"

print(men1)
print(men2)
```

```
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Murtaza Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Murtaza Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
```

Even though we made a copy of men1 dictionary using copy method and changed only men2 dictionary, but contents of men1 are still changed when we change men2.

This is because copy method still makes a shallow copy of the dictionaries (1,2) which are inside the dictionary (men1).

Same is true for *list* in the dictionaries.

```
books1 = {"AlSadr": ["Our Philosophy", "Our Economy"],
          "Mutahri": ["Divine Justice", "Man and Destiny"]}

books2 = books1.copy()

books2["Mutahri"][1] = "The goal of life"
print(books1)
print(books2)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'The goal of␣
↪life']}
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'The goal of␣
↪life']}
```

How to copy a dictionary which may contain several dictionaries?

we can make use of copy by iterating over dictionary

```
import copy

men1 = {1: {"name": "Baqir -al- Sadr", "born": 1935, "citizenship": "Iraq", "died": 1980}
↪,
        2: {"name": "Mutahri", "born": 1919, "citizenship": "Iran", "died": 1979}}

men2 = copy.copy(men1)

men2[2]["name"] = "Murtaza Mutahri"

print(men1)
print(men2)
```

```
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Murtaza Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Murtaza Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
```

if we iterate through each key, value pair of dictionary and copy it individually, we can avoid this shallow copying

```
def copy_dict(d: dict) -> dict:
    """makes deepcopy of a dictionary without cloning it"""
    assert isinstance(d, dict)

    new_dict = {}
    for k, v in d.items():
        new_dict[k] = copy.copy(v)
    return new_dict


men1 = {1: {"name": "Baqir -al- Sadr", "born": 1935, "citizenship": "Iraq", "died": 1980}
↪,
        2: {"name": "Mutahri", "born": 1919, "citizenship": "Iran", "died": 1979}}

men2 = copy_dict(men1)


men2[2]["name"] = "Murtaza Mutahri"

print(men1)
print(men2)
```

```
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
{1: {'name': 'Baqir -al- Sadr', 'born': 1935, 'citizenship': 'Iraq', 'died': 1980}, 2: {
↪'name': 'Murtaza Mutahri', 'born': 1919, 'citizenship': 'Iran', 'died': 1979}}
```

but what if dictionary inside the dictionary further contains dictionaries

```
men1 = {1: {"iraq": {'person1': {'name': 'sadr'}, 'person2': {'name': 'hakim'}},
            "iran": {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}

men2 = copy_dict(men1)

men2[1]["iraq"]['person1']['name'] = "baqir al sadr"

print(men1)
print(men2)
```

```
{1: {'iraq': {'person1': {'name': 'baqir al sadr'}, 'person2': {'name': 'hakim'}}, 'iran
↪': {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
{1: {'iraq': {'person1': {'name': 'baqir al sadr'}, 'person2': {'name': 'hakim'}}, 'iran
↪': {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
```

although we changed `name` of `person1` in `men2` but it is also changed in `men1`.

we can achieve this by calling the parent function again every time the value is a dictionary i.e. calling `copy_dict` function inside `copy_dict` function.

```
def copy_dict(d: dict) -> dict:
    """makes deepcopy of a dictionary without cloning it"""
    assert isinstance(d, dict)

```

(continues on next page)

(continued from previous page)

```
    new_dict = {}
    for k, v in d.items():
        if isinstance(v, dict):
            new_dict[k] = copy_dict(v)
        else:
            new_dict[k] = copy.copy(v)
    return new_dict


men1 = {1: {"iraq": {'person1': {'name': 'sadr'}, 'person2': {'name': 'hakim'}},
            "iran": {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}

men2 = copy_dict(men1)

men2[1]["iraq"]['person1']['name'] = "baqir al sadr"

print(men1)
print(men2)
```

```
{1: {'iraq': {'person1': {'name': 'sadr'}, 'person2': {'name': 'hakim'}}, 'iran': {
↪'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
{1: {'iraq': {'person1': {'name': 'baqir al sadr'}, 'person2': {'name': 'hakim'}}, 'iran
↪': {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
```

However, there is simpler solution to this problem. Instead of writting a function like *copy_dict*, which copies each object from dictionary one by one, we can simply use `deepcopy` function from `copy` module.

```
from copy import deepcopy

men1 = {1: {"iraq": {'person1': {'name': 'sadr'}, 'person2': {'name': 'hakim'}},
            "iran": {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}

men2 = deepcopy(men1)

men2[1]["iraq"]['person1']['name'] = "baqir al sadr"

print(men1)
print(men2)
```

```
{1: {'iraq': {'person1': {'name': 'sadr'}, 'person2': {'name': 'hakim'}}, 'iran': {
↪'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
{1: {'iraq': {'person1': {'name': 'baqir al sadr'}, 'person2': {'name': 'hakim'}}, 'iran
↪': {'person1': {'name': 'mutahri'}, 'person2': {'name': 'shariati'}}}}
```

update

This method updates an existing dictionary. %%

```
books = {"AlSadr": ["Our Philosophy", "Our Economy"],
         "Mutahri": ["Divine Justice", "Man and Destiny"]
         }

new_books = {"Legenhausen": ["Religious pluralism", "Hegel's ethics"]}
```

The method does not return anything. Only the original dictionary is changed.

```
books.update(new_books)

print(books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny'], 'Legenhausen': ['Religious pluralism', "Hegel's ethics"]}
```

Merging dictionaries

The update merges one dictionary into other. If we want to keep both dictionaries intact and create a new one by merging them together, we can do this as following (starting from python 3.5)

```
old_books = {"AlSadr": ["Our Philosophy", "Our Economy"],
             "Mutahri": ["Divine Justice", "Man and Destiny"]
             }

new_books = {"Legenhausen": ["Religious pluralism", "Hegel's ethics"]}

books = {**old_books, **new_books}

print(books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny'], 'Legenhausen': ['Religious pluralism', "Hegel's ethics"]}
```

We can verify that *old_books* and *new_books* dictionaries are intact.

```
print(old_books)

print(new_books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny']}
{'Legenhausen': ['Religious pluralism', "Hegel's ethics"]}
```

We can even provide a new key value pair.

```
books = {**old_books, "Iqbal": "reconstruction", **new_books}

print(books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny'], 'Iqbal': 'reconstruction', 'Legenhausen': ['Religious pluralism', "Hegel's␣
↪ethics"]}
```

The double asterisk **, in fact, just unpacks the dictionary into key value pairs and then we construct a new dictionary by putting the unpacked key value pairs inside curly brackets "{}".

```
print({'x': 1, **{'y': 2}})
```

```
{'x': 1, 'y': 2}
```

For backup compatability, we better use the `update` method that can run on versions before 3.5 as follows

```
books = old_books.copy()
books.update(new_books)
print(books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny'], 'Legenhausen': ['Religious pluralism', "Hegel's ethics"]}
```

We can also merge two dictionaries with another method.

```
old_books = {"AlSadr": ["Our Philosophy", "Our Economy"],
             "Mutahri": ["Divine Justice", "Man and Destiny"]}

new_books = {"Legenhausen": ["Religious pluralism", "Hegel's ethics"]}

books = dict(list(old_books.items()) + list(new_books.items()))
print(books)
```

```
{'AlSadr': ['Our Philosophy', 'Our Economy'], 'Mutahri': ['Divine Justice', 'Man and␣
↪Destiny'], 'Legenhausen': ['Religious pluralism', "Hegel's ethics"]}
```

Question: Write code to print the value of second key of the following dictionary i.e. "Hassan".

```
x = {1: "ali", 2: "hassan", 3: "hussain"}
```

Question: Write code to tell the date of birth and death of the *Ali* from the following dictionary.

```
x = {"Ali": {"born": 600, "died": 661}, "Hassan": {"born": 625, "died": 670}, "Hussain":
↪{"born": 626, "died": 680}}
```

Question: What will be output of following code?

```
x = {1: "ali", 2: "hassan", 3: "hussain"}
y = {1: "ali", 2: "hassan", 3: "hussain", 4: "Ali"}
print(x.get(4, y.get(4))
```

Question: Change the contents of dictionary *y* in such a way the following code throws `KeyError`

```
x = {1: "ali", 2: "hassan", 3: "hussain"}
y = ??
print(x.get(4, y.get(4))  # should throw KeyError
```

Total running time of the script: (0 minutes 0.023 seconds)

1.7 sets

Important: This lesson is still under development.

A set is a collection of objects just like lists with the exception that it is unordered, does not contain same objects more than once, and can not contain immutable objects like lists.

A set can be created from an existing sequence object such as a string, list or tuple.

```
urdu = set("National language of Pakistan")

print(type(urdu))

print(urdu)
```

```
<class 'set'>
{'i', 'a', ' ', 'n', 'l', 'f', 'g', 'u', 'e', 't', 'k', 's', 'o', 'P', 'N'}
```

```
pak_langs = set(["Balochi", "Barohi", "Sindhi", "Balti"])
print(pak_langs)
```

```
{'Balochi', 'Barohi', 'Sindhi', 'Balti'}
```

If our sequence contains repeating objects, only one instance of those repeating objects will be included in the list.

```
pak_langs = set(("Balochi", "Barohi", "Sindhi", "Balti", "Balochi"))
print(pak_langs)
```

```
{'Balochi', 'Barohi', 'Sindhi', 'Balti'}
```

Although, we can create sets from lists, but a set can not contain a list as an object.

```
pak_langs = set((("Balochi", "Barohi"), ("punbabi", "siraiki")))
print(pak_langs)
```

```
{('punbabi', 'siraiki'), ('Balochi', 'Barohi')}
```

```
# uncomment following line
# pak_langs = set((["Balochi", "Barohi"], ["punbabi", "siraiki"]))
print(pak_langs)
```

```
{('punbabi', 'siraiki'), ('Balochi', 'Barohi')}
```

In second case above, we want our set to have two lists as objects, so the error was prompted. Sets are mutable i.e. they can be changed. We can add new objects in sets as following

```
pak_langs = set(["Balochi", "Barohi", "Sindhi"])
pak_langs.add("Pashto")
print(pak_langs)
```

```
{'Balochi', 'Barohi', 'Sindhi', 'Pashto'}
```

There are immutable sets as well with the name *frozenset*.

```
balochistan_langs = frozenset(["Balochi", "Barohi", "Pashto"])

# uncomment following line
# balochistan_langs.add("punjabi")
```

```
# Operations on sets
```

adding elements

We saw, how to add objects in sets with the method *add*. We can not violate aforementioned rules using *add* method.

```
imperialists = {"bbc", "cnn"}

# uncomment following line
# imperialists.add(["voa","dw"])  # TypeError
```

```
imperialists.add('bbc')
print(imperialists)
```

```
{'bbc', 'cnn'}
```

```
imperialists.update(["voa","dw"])
print(imperialists)
```

```
{'bbc', 'voa', 'cnn', 'dw'}
```

```
imperialists = {"bbc", "cnn"}

# uncomment following line
# imperialists.update([["voa","dw"]])  # TypeError
print(imperialists)
```

```
{'bbc', 'cnn'}
```

| operator can also be used to add/concatenate two sets

```
imperialists = {"bbc", "cnn"}

imperialists | {"voa", "dw"}
```

```
{'bbc', 'voa', 'cnn', 'dw'}
```

```
imperialists = {"bbc", "cnn"}

imperialists |= {"voa", "dw"}
```

(continues on next page)

(continued from previous page)

```
print(imperialists)
```

```
{'bbc', 'voa', 'cnn', 'dw'}
```

clear

We can clear the contents of a set by using the method *clear* on a set.

```
dakus = {"musharaf", "nawaz", "benazir"}

dakus.clear()  # after NRO (https://en.wikipedia.org/wiki/National_Reconciliation_
↪Ordinance)
print(dakus)
```

```
set()
```

Copy

The assignment operation = does not create a new set.

```
more_dakus = {"pervaiz elahi", "altaf husain"}
dakus_backup = more_dakus
more_dakus.clear()
print(dakus_backup)
```

```
set()
```

copy method creates a shallow copy

```
more_dakus = {"pervaiz elahi", "altaf husain"}
dakus_backup = more_dakus.copy()
more_dakus.clear()
print(dakus_backup)
```

```
{'altaf husain', 'pervaiz elahi'}
```

```
imperialists = {"BBC", "CNN", "VOA"}

more_imperialists = imperialists.copy()

more_imperialists.add("DW")

print(imperialists)

print(more_imperialists)
```

```
{'CNN', 'BBC', 'VOA'}
{'CNN', 'DW', 'BBC', 'VOA'}
```

difference

```
pml_q = {"zafrullah jamali", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_n = {"choi Nisar" , "umar ayyub", "khawaja Asif"}
pti = {"firdows ashiq", "umar ayyub", "asad umar", "fawad hussain"}

pml_q.difference(pti)
```

```
{'pervaiz elahi', 'zafrullah jamali'}
```

```
lotas_2013 = pml_q.difference(pml_q.difference(pml_n))
print(lotas_2013)
```

```
{'umar ayyub'}
```

We can also make use of - operator

```
print(pml_q - pti)
```

```
{'pervaiz elahi', 'zafrullah jamali'}
```

difference_update

This makes change in original set. similar to *x-y* with the exception that *x* is itself changed.

```
pml_q = {"zafrullah jamali", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_n = {"choi Nisar" , "umar ayyub", "khawaja Asif"}
pti = {"firdows ashiq", "umar ayyub", "asad umar", "fawad hussain"}

pml_q.difference_update(pml_n)

print(pml_q)
```

```
{'zafrullah jamali', 'fawad hussain', 'pervaiz elahi'}
```

```
pml_q.difference_update(pti)

print(pml_q)
```

```
{'pervaiz elahi', 'zafrullah jamali'}
```

discard

removes an element from set if it is present.

```
pml_q = {"zafrullah jamali", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_q.discard("zafrullah jamali")
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'pervaiz elahi'}
```

```
pml_q.discard("choi nisar")
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'pervaiz elahi'}
```

ferdows ashiq is not present in set *musharaf* but using *discard* did not raise an error.

```
## ``remove``
# Same as `discard` with the exception that an error is raised if the object is
# not present in set.
```

```
pml_q = {"zafrullah jamali", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_q.remove("zafrullah jamali")
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'pervaiz elahi'}
```

```
# uncomment following line
# pml_q.remove("choi nisar")  # KeyError
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'pervaiz elahi'}
```

pop

```
pml_q = {"firdows ashiq", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_q.pop()
print(pml_q)
```

```
{'umar ayyub', 'firdows ashiq', 'pervaiz elahi'}
```

```
pml_q.pop()
print(pml_q)
```

```
{'firdows ashiq', 'pervaiz elahi'}
```

Running the above cell multiple times will eventually raise an error when the set becomes empty.

union

```
pml_q = {"firdows ashiq", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pml_n = {"choi Nisar" , "umar ayyub", "khawaja Asif"}

pml_q.union(pml_n)
```

```
{'khawaja Asif', 'fawad hussain', 'umar ayyub', 'firdows ashiq', 'choi Nisar', 'pervaiz␣
↪elahi'}
```

```
print(pml_q | pml_n)
```

```
{'khawaja Asif', 'fawad hussain', 'umar ayyub', 'firdows ashiq', 'choi Nisar', 'pervaiz␣
↪elahi'}
```

```
## `intersection`
```

```
pml_q = {"firdows ashiq", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pti = {"firdows ashiq", "umar ayyub", "asad umar", "fawad hussain"}

pml_q.intersection(pti)
```

```
{'fawad hussain', 'umar ayyub', 'firdows ashiq'}
```

We can also use & operator

```
print(pml_q & pti)
```

```
{'fawad hussain', 'umar ayyub', 'firdows ashiq'}
```

The original set *pml_q* remains unchanged.

```
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'firdows ashiq', 'pervaiz elahi'}
```

However, if we use *intersection_update*, the original set is changed

```
pml_q.intersection_update(pti)
print(pml_q)
```

```
{'fawad hussain', 'umar ayyub', 'firdows ashiq'}
```

If we want to find out intersection between multiple sets, we can do it as following.

```
pml_q = {"firdows ashiq", "fawad hussain", "pervaiz elahi", "umar ayyub"}
pti = {"firdows ashiq", "umar ayyub", "asad umar", "fawad hussain"}
pml_n = {"choi Nisar" , "umar ayyub", "khawaja Asif"}

sets = [pml_q, pml_n, pti]
set.intersection(*sets)
```

```
{'umar ayyub'}
```

or

```
sets = [pml_n, pti]
pml_q.intersection(*sets)
```

```
{'umar ayyub'}
```

So we can say that umar ayyub is the most consistent turncoat.

isdisjoint

returns *True* if the intersection of two sets is not empty set.

```
ppp = {"firdows ashiq", "fawad hussain", "Amin Faheem", "umar ayyub"}
pti = {"firdows ashiq", "umar ayyub", "asad umar", "fawad hussain"}
ji = {"liaquat baloch", "siraj ul haq", "munawar hasan"}

ppp.isdisjoint(ji)
```

```
True
```

```
ppp.isdisjoint(pti)
```

```
False
```

```
# `issubset`
# ------------
# ``<`` is used for proper subset and ``<=`` is used for subset checking.
```

```
pml_n = {"nawaz", "shahbaz", "pervaiz elahi", "mushahid husain"}
pml_q = {"pervaiz elahi", "mushahid husain"}

pml_q.issubset(pml_n)
```

```
True
```

```
print(pml_q <= pml_n)
```

```
True
```

```
print(pml_q < pml_q)
```

```
False
```

issuperset

> is used for proper superset and >= is used for superset checking.

```
pml_n = {"nawaz", "shahbaz", "pervaiz elahi", "mushahid husain"}
pml_q = {"pervaiz elahi", "mushahid husain"}

pml_n.issuperset(pml_q)
```

```
True
```

```
print(pml_n >= pml_q)
```

```
True
```

```
print(pml_n > pml_n)
```

```
False
```

```
# Since sets are unordered, the operation ``in`` is faster when applied to
# sets as compared to lists.
```

```
print("nawaz" in pml_n)
```

```
True
```

```
print("nawaz" not in pml_q)
```

```
True
```

Total running time of the script: (0 minutes 0.015 seconds)

1.8 conditional statements

Important: This lesson is still under development.

The basic syntax of if statement in python is:

```
if (condition):
    do something
```

```
name = "zardari"

if name == 'zardari':
    print('chor')
```

```
chor
```

```
name = 'Zardari'
if name == 'zardari' or name == 'Zardari':
    print('chor')
```

```
chor
```

```
name = 'nawaz'
if name == 'zardari' or name == 'Zardari':
    print('chor')
if name == 'nawaz' or name == 'Nawaz':
    print('chor')
```

```
chor
```

Similarly the syntax for *if* and *elif* statement is

```
if (condition):
  do something
elif (condition):
  do something
else:
  do something
```

```
name = 'edhi'
if name == 'zardari' or name == 'Zardari':
    print(name, ' is chor')
elif name == 'nawaz' or name == 'Nawaz':
    print(name, ' is chor')
else:
    print(name, ' is not chor')
```

```
edhi  is not chor
```

`elif` vs multiple `if`

Multiple ifs means, the all the ifs will be checked, while with elif, the code will stop if one of the if is True.

```
age = 14

if age < 16:
    print("You are a child")
if age >= 16: #Greater than or equal to
    print("You are an adult")
else:   #Handle all cases where 'age' is negative
    print("The age must be a positive integer!")
```

```
You are a child
The age must be a positive integer!
```

```
age = 14

if age < 16:
       print("You are a child")
elif age > 16:
       print("You are an adult")
else:   #Handle all cases where 'age' is negative
       print("The age must be a positive integer!")
```

```
You are a child
```

in

We can use *in* statement to compare a variable against multiple variables.

```
name = 'zardari'
if name in ['zardari', 'nawaz', 'chaudhrys', 'mqm']:
    print(name, ' was democratic thug')
elif name in ['zia', 'yahya', 'musharaf', 'ayub']:
    print(name, ' was non-democratic thug')
else:
    print(name, ' is not chor')
```

```
zardari  was democratic thug
```

```
name = 'Zia'
if name.upper() in ['ZARDARI', 'NAWAZ', 'CHAUDHRYS', 'MQM']:
    print(name, ' was a democratic chor')
elif name.upper() in ['ZIA', 'YAHYA', 'MUSHARAF', 'AYUB']:
    print(name, ' was a non-democratic is chor')
else:
    print(name, ' is not chor')
```

```
Zia  was a non-democratic is chor
```

comparing numbers

```
year = 2009

if 2007>year>2000:
    print('Non-democratic thug ruled Pakistan')
elif 2020>year>=2007:
    print('democratic thug ruled Pakistan')
else:
    print('not considering')
```

```
democratic thug ruled Pakistan
```

```
year = 2012

if 2007>year>2000:
    print('Non-democratic thug ruled Pakistan')
elif 2020>year>=2007:

    if 2013>year>=2007:
        print('democratic thug zardari ruled Pakistan')
    elif 2018>year>=2013:
        print('democratic thug Nawaz ruled Pakistan')
    else:
        print('It seems the ruler is incapable')
else:
    print('not considering')
```

```
democratic thug zardari ruled Pakistan
```

One liner

```
day = "14 aug"

if day == '14 aug':  print('This is independence day not partition day.')
```

```
This is independence day not partition day.
```

```
oil = True
us_presence = 1 if oil else 0

print(us_presence)
```

```
1
```

We can use such one liners to set default values to a variable.

```
human = {"arms": 2,
         "legs": 2,
         "head": 1}

default_age = 14

age = human["age"] if "age" in human else default_age
print(age)
```

```
14
```

```
provinces = 4
capital = "Kathmandu"
pm = "unknown"

if provinces == 4 and capital == "Islamabad" or pm == "Imran Khan":
    print("This is Pakistan")
```

```
provinces = 4
capital = "unknown"
pm = "Imran Khan"

if provinces == 4 and capital == "Islamabad" or pm == "Imran Khan":
    print("This is Pakistan")
```

```
This is Pakistan
```

if not vs !=

Inner working is different but the output is same however `not` is preferred.

```
day = "Thursday"
if day != 'Friday':
    print('no jumma prayer')
```

```
no jumma prayer
```

```
day = "Thursday"
if not day == 'Friday':
    print('no jumma prayer')
```

```
no jumma prayer
```

```
if day == 'Friday':
    pass
else:
    print("no jumma prayer")
```

```
no jumma prayer
```

any vs all

```
a = [False, 2>4, 2!=1]
if any(a):
    print('go ahead')
```

```
go ahead
```

```
a = [False, 2>4, 2!=1]
if all(a):
    print('go ahead')
```

Question

What will be the output of following code?

```
a = [True, 2>4, 2!=1]
if any(a):
    print('go ahead')
```

Total running time of the script: (0 minutes 0.007 seconds)

1.9 while loops

Important: This lesson is still under development.

Suppose we have a list of names of people who have ruled Pakistan with one name in it who has not been the ruler and we want to find out at which position the name of this person is located. One way to solve this problem is to use while loops.

```
looters = ['sikandar mirza', 'ayub khan', 'yahya khan', 'zulfiqar bhutto', 'shahid afridi
↪',
           'zia-ul-haq', 'benazir', 'nawaz sharif', 'musharaf',   'zardari']

acc_id = 0
while acc_id<len(looters):   # this condition must become False at some point.
    if looters[acc_id] == 'shahid afridi':
        print('Found a normal person at position', acc_id)
    else:
        print(looters[acc_id], 'was a thug')
    acc_id += 1
```

```
sikandar mirza was a thug
ayub khan was a thug
yahya khan was a thug
zulfiqar bhutto was a thug
Found a normal person at position 4
zia-ul-haq was a thug
benazir was a thug
nawaz sharif was a thug
musharaf was a thug
zardari was a thug
```

The basic syntax of while statement in python is:

```
while (condition):
    do something
```

The condition after `while` must become *False* after some time otherwise the loop will continue indefinitely. Consider not increasing the value of *acc_id* in upper example and the print statement will continue forever until we have to stop it forcefully by terminating the program. (In case you do do this, you can stop this by going to *Runtime –> Interrupt execution*.)

while with else

```
while condition:
  do something
```

(continues on next page)

(continued from previous page)

```
else:
  do something at last
```

```
looters = ['sikandar mirza', 'ayub khan', 'yahya khan', 'zulfiqar bhutto', 'shahid afridi
↪',
           'zia-ul-haq', 'benazir', 'nawaz sharif', 'musharaf', 'zardari']

acc_id = 0
while acc_id<len(looters):
    if looters[acc_id] == 'shahid afridi':
        print('Found a normal person at position', acc_id)
    else:
        print(looters[acc_id], 'was a thug')
    acc_id += 1
else:
    print("Search finished from the whole list of 'looters'")
```

```
sikandar mirza was a thug
ayub khan was a thug
yahya khan was a thug
zulfiqar bhutto was a thug
Found a normal person at position 4
zia-ul-haq was a thug
benazir was a thug
nawaz sharif was a thug
musharaf was a thug
zardari was a thug
Search finished from the whole list of 'looters'
```

```
looters = ['sikandar mirza', 'ayub khan', 'yahya khan', 'zulfiqar bhutto', 'shahid afridi
↪',
           'zia-ul-haq', 'benazir', 'nawaz sharif', 'musharaf', 'zardari']

acc_id = 0
while acc_id<len(looters):
    if looters[acc_id] == 'shahid afridi':
        print('Found a normal person at position ', acc_id)
    else:
        print(looters[acc_id], 'was a thug')
    acc_id += 1

print("Search finished from the whole list of 'looters'")
```

```
sikandar mirza was a thug
ayub khan was a thug
yahya khan was a thug
zulfiqar bhutto was a thug
Found a normal person at position  4
zia-ul-haq was a thug
benazir was a thug
nawaz sharif was a thug
```

(continues on next page)

(continued from previous page)

```
musharaf was a thug
zardari was a thug
Search finished from the whole list of 'looters'
```

We may think *what is the use of else statement*? Of course we can achieve same thing by just placing the code inside *else* statement, without making use of *else*.

Wouldn't it be better if we just stop the search after we found the thief. Here comes the benefit of *break* statement along with *else* keyword.

```
looters = ['sikandar mirza', 'ayub khan', 'yahya khan', 'zulfiqar bhutto', 'shahid afridi
↪',
           'zia-ul-haq', 'benazir', 'nawaz sharif', 'musharaf', 'zardari']

acc_id = 0
while acc_id<len(looters):
    if looters[acc_id] == 'shahid afridi':
        print('Found a normal person at position', acc_id, '. No need to continue␣
↪searching anymore')
        break
    else:
        print(looters[acc_id], 'was a thug')
    acc_id += 1
else:
    print("Search finished from the whole list of 'looters'")
```

```
sikandar mirza was a thug
ayub khan was a thug
yahya khan was a thug
zulfiqar bhutto was a thug
Found a normal person at position 4 . No need to continue searching anymore
```

Question What is the output of the following code

```
x = 0
y = 5
while x+y<10:
    x += 1
    print(x)
```

Total running time of the script: (0 minutes 0.003 seconds)

1.10 for loops

This lesson introduces `for` loops in python.

Important: This lesson is still under development.

Just like while loops, `for` loops allow an instruction to be executed a certain number of times. How many times? It depends upon iterator. As per wikipedia 11 people have received Nishan-e-Haider[1] award in Pakistan. Let's say we

[1] https://en.wikipedia.org/wiki/Nishan-e-Haider

want to iterate over this list.

```
NH_receivers = ['Saif Ali Janjua', 'Muhammad Sarwar', 'Tufail Muhammad',
                'Aziz Bhatti', 'Rashid Minhas', 'Muhammad Akran',
                'Shabbir Sharif', 'Muhammad Husain Janjua', 'Muhammad Mahfuz',
                'Sher Khan', 'Lalak Jan']

for shaheed in NH_receivers:
    print(shaheed)
```

```
Saif Ali Janjua
Muhammad Sarwar
Tufail Muhammad
Aziz Bhatti
Rashid Minhas
Muhammad Akran
Shabbir Sharif
Muhammad Husain Janjua
Muhammad Mahfuz
Sher Khan
Lalak Jan
```

In above example, a list is acting as an iterator. In fact it can be a tuple, string or any other sequence. The basic syntax of for loop in python is:

```
for variable in sequence:
    do something
```

The *variable* in above syntax is assigned a new new value from *sequence* upon every iteration.

```
for podcast in ("Hujjat", "The east is the podcast", "Philosophise this"):
    print("I am listening to: ", podcast)
```

```
I am listening to:  Hujjat
I am listening to:  The east is the podcast
I am listening to:  Philosophise this
```

The variable *podcast* is assigned a new value from the sequence (tuple in this case).

Question: Print first 5 natural numbers using for loop.

We can also run a for loop on items of a dictionary. If we want to iterate over both keys and values of a dictions, we would do as below

```
scholars = {
    "Baqir al sadr": 1980,
    "Murtaza Mutahri": 1979,
    "Allama Iqbal": 1938,
    "Jamal ul din Afghani": 1897,
    "Ali Shariati": 1977,
    "Moh Husain Tabatabai": 1981
}

for scholar, date_of_death in scholars.items():
    print(scholar, "died in ", date_of_death)
```

```
Baqir al sadr died in  1980
Murtaza Mutahri died in  1979
Allama Iqbal died in  1938
Jamal ul din Afghani died in  1897
Ali Shariati died in  1977
Moh Husain Tabatabai died in  1981
```

Question:

```
for a in range(0, 10):
    if a>5:
        a = a + 100
```

print(a) What will be the output of above code?

for with else

Just like *while* loops, the code under else gets executed if everything goes well within *for* loop. If let's say, we are running for loop to search an item in the iterator, and when we find the item, there is not point in continuing the *for* loop further. Hence we *break* out of for loop. In such a case code under *else* will not be executed.

```
for variable in sequence:
    do something
else:
    do at last
```

```
for shaheed in NH_receivers:
    if shaheed == 'Rashid Minhas':
        print("Person from air force found")
        break
else:
    print('Search completed')
```

```
Person from air force found
```

We can create a simple sequence using `range` function

```
range(5)
```

```
range(0, 5)
```

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

We can alo iterate over a list using range.

```
for i in range(5):
    print(NH_receivers[i])
```

```
Saif Ali Janjua
Muhammad Sarwar
Tufail Muhammad
Aziz Bhatti
Rashid Minhas
```

However, the above example is not a very clever approach as we could have simple done `for i in NH_receivers`. A more useful example would be:

```
for i in range(2, 5):
    print(NH_receivers[i])
```

```
Tufail Muhammad
Aziz Bhatti
Rashid Minhas
```

We can also include step argument in range, which decides how big the jump/step we want to have in our iterator. In this way we can skip every nth value in a squence/iterator.

```
for i in range(2, 8, 2):
    print(NH_receivers[i])
```

```
Tufail Muhammad
Rashid Minhas
Shabbir Sharif
```

We can also go backwards in the iterator

```
for i in range(8, 2, -1):
    print(NH_receivers[i])
```

```
Muhammad Mahfuz
Muhammad Husain Janjua
Shabbir Sharif
Muhammad Akran
Rashid Minhas
Aziz Bhatti
```

```
for i in range(8, 0, -2):
    print(NH_receivers[i])
```

```
Muhammad Mahfuz
Shabbir Sharif
Rashid Minhas
Tufail Muhammad
```

nested for loops

```
for name in NH_receivers:
    for char in name:
        print(char)
```

```
S
a
i
f

A
l
i

J
a
n
j
u
a
M
u
h
a
m
m
a
d

S
a
r
w
a
r
T
u
f
a
i
l

M
u
h
a
m
m
a
d
A
```

(continues on next page)

(continued from previous page)

```
z
i
z

B
h
a
t
t
i
R
a
s
h
i
d

M
i
n
h
a
s
M
u
h
a
m
m
a
d

A
k
r
a
n
S
h
a
b
b
i
r

S
h
a
r
i
f
M
```

(continues on next page)

(continued from previous page)

```
u
h
a
m
m
a
d

H
u
s
a
i
n

J
a
n
j
u
a
M
u
h
a
m
m
a
d

M
a
h
f
u
z
S
h
e
r

K
h
a
n
L
a
l
a
k

J
```

(continues on next page)

(continued from previous page)

```
a
n
```

accessing index

If we want to access index itself, we can do this by using `enumerate`.

```
for index, item in enumerate(NH_receivers, start=0):  # default value of start is zero.
    print(item, ' is Nishan - Haider receiver number ', index)
```

```
Saif Ali Janjua  is Nishan - Haider receiver number  0
Muhammad Sarwar  is Nishan - Haider receiver number  1
Tufail Muhammad  is Nishan - Haider receiver number  2
Aziz Bhatti  is Nishan - Haider receiver number  3
Rashid Minhas  is Nishan - Haider receiver number  4
Muhammad Akran  is Nishan - Haider receiver number  5
Shabbir Sharif  is Nishan - Haider receiver number  6
Muhammad Husain Janjua  is Nishan - Haider receiver number  7
Muhammad Mahfuz  is Nishan - Haider receiver number  8
Sher Khan  is Nishan - Haider receiver number  9
Lalak Jan  is Nishan - Haider receiver number  10
```

which is equivalent to

```
index = 0
for item in NH_receivers:
    print(item, ' is Nishan - Haider receiver number ', index)
    index += 1
```

```
Saif Ali Janjua  is Nishan - Haider receiver number  0
Muhammad Sarwar  is Nishan - Haider receiver number  1
Tufail Muhammad  is Nishan - Haider receiver number  2
Aziz Bhatti  is Nishan - Haider receiver number  3
Rashid Minhas  is Nishan - Haider receiver number  4
Muhammad Akran  is Nishan - Haider receiver number  5
Shabbir Sharif  is Nishan - Haider receiver number  6
Muhammad Husain Janjua  is Nishan - Haider receiver number  7
Muhammad Mahfuz  is Nishan - Haider receiver number  8
Sher Khan  is Nishan - Haider receiver number  9
Lalak Jan  is Nishan - Haider receiver number  10
```

This is another way to keep track that how many times the loop has been executed.

Question:

```
for i in enumerate(range(5)):
    pass
```

print(type(i))

```
# What will be the output of above code?
```

Iterating over more than one sequences

If we want to iterate over more than one sequences, we can do this using built-in function `zip`.

```
scholars = ['Baqir al sadr', 'Murtaza Mutahri', 'Allama Iqbal', 'Jamal ul din Afghani',
            'Ali Shariati', 'Moh Husain Tabatabai']
date_of_death = [1980, 1979, 1938, 1897, 1977, 1981]

for scholar, dod in zip(scholars, date_of_death):
    print(scholar, ' died in year ', dod)
```

```
Baqir al sadr  died in year  1980
Murtaza Mutahri  died in year  1979
Allama Iqbal  died in year  1938
Jamal ul din Afghani  died in year  1897
Ali Shariati  died in year  1977
Moh Husain Tabatabai  died in year  1981
```

```
date_of_birth = [1935, 1919, 1877, 1838, 1933, 1904]
for scholar, dod, dob in zip(scholars, date_of_death, date_of_birth):
    print(scholar, ' was born in ', dob, ' and died in year ', dod)
```

```
Baqir al sadr  was born in  1935  and died in year  1980
Murtaza Mutahri  was born in  1919  and died in year  1979
Allama Iqbal  was born in  1877  and died in year  1938
Jamal ul din Afghani  was born in  1838  and died in year  1897
Ali Shariati  was born in  1933  and died in year  1977
Moh Husain Tabatabai  was born in  1904  and died in year  1981
```

But what if lengths of lists are not equal?

```
scholars = ['Baqir al sadr', 'Murtaza Mutahri', 'Allama Iqbal', 'Jamal ul din Afghani',
            'Ali Shariati', 'Moh Husain Tabatabai']
date_of_death = [1980, 1979, 1938, 1897, 1977, 1981, 1989]

print(len(scholars), len(date_of_death))

for scholar, dod in zip(scholars, date_of_death):
    print(scholar, ' died in year ', dod)
```

```
6 7
Baqir al sadr  died in year  1980
Murtaza Mutahri  died in year  1979
Allama Iqbal  died in year  1938
Jamal ul din Afghani  died in year  1897
Ali Shariati  died in year  1977
Moh Husain Tabatabai  died in year  1981
```

Simple `zip` will iterate over the point when all lists are equal and ignore if any sequence is larger than the others. If we want to iterate until the longest sequence, we have to use `zip_longest` from `itertools`

```
from itertools import zip_longest

scholars = ['Baqir al sadr', 'Murtaza Mutahri', 'Allama Iqbal', 'Jamal ul din Afghani',
            'Ali Shariati', 'Moh Husain Tabatabai']
date_of_death = [1980, 1979, 1938, 1897, 1977, 1981, 1989]

for scholar, dod in zip_longest(scholars, date_of_death):
    print('name: ', scholar, ' date of death: ', dod)
```

```
name:  Baqir al sadr  date of death:  1980
name:  Murtaza Mutahri  date of death:  1979
name:  Allama Iqbal  date of death:  1938
name:  Jamal ul din Afghani  date of death:  1897
name:  Ali Shariati  date of death:  1977
name:  Moh Husain Tabatabai  date of death:  1981
name:  None  date of death:  1989
```

Notice the last printed line. If we want to access the previous and next value during iteration, we must start from 1 and end before last item in order to print correct values

```
for i in range(1, len(NH_receivers) - 1):
    print(NH_receivers[i], ' came before ', NH_receivers[i + 1], ' and after ', NH_
↪receivers[i - 1])
```

```
Muhammad Sarwar  came before  Tufail Muhammad  and after  Saif Ali Janjua
Tufail Muhammad  came before  Aziz Bhatti  and after  Muhammad Sarwar
Aziz Bhatti  came before  Rashid Minhas  and after  Tufail Muhammad
Rashid Minhas  came before  Muhammad Akran  and after  Aziz Bhatti
Muhammad Akran  came before  Shabbir Sharif  and after  Rashid Minhas
Shabbir Sharif  came before  Muhammad Husain Janjua  and after  Muhammad Akran
Muhammad Husain Janjua  came before  Muhammad Mahfuz  and after  Shabbir Sharif
Muhammad Mahfuz  came before  Sher Khan  and after  Muhammad Husain Janjua
Sher Khan  came before  Lalak Jan  and after  Muhammad Mahfuz
```

We can change loop variable inside the loop. However, this is not a good practice and is prone to bugs. Let's say we want to iterate over sequence from 0 to 10 i.e *0,1,2,3,4,5,6,7,8,9* but after *5* we want to jump to *8* and then carry on. It means we should change the loop variable inside the loop. A naive mind might thing following approach would work %%

```
for i in range(10):
    if i == 5:
        i += 3
    print(i)
```

```
0
1
2
3
4
8
6
7
```

(continues on next page)

(continued from previous page)

```
8
9
```

But we had wished the output as *0,1,2,3,4,5,8,9*. The reasons is, for loop iterated over *0,1,2,3,4,5,6,7,8,9* and if we changed current value of *i*, it will not affect next value of *i* at next iteration. We have to use *while* loop in such a scenario.

```
i = 0
while i < 10:
    if i == 5:
        i += 3
    print(i)
    i += 1
```

```
0
1
2
3
4
8
9
```

If we have a list of lists, and we want to flatten all the elements of that list into one list, we can use a nested for loop to achieve this.

```
prime_ministers = [['zafrullah jamali', 'chaudhry shujaat', 'shaukat aziz'],
                   ['yousuf raza gilani', 'raja pervaiz ashraf'],
                   ['nawaz sharif', 'shahid khaqan']]

print(len(prime_ministers))

all_pms = []
for era in prime_ministers:
    for pm in era:
        all_pms.append(pm)

print(all_pms)
```

```
3
['zafrullah jamali', 'chaudhry shujaat', 'shaukat aziz', 'yousuf raza gilani', 'raja␣
↪pervaiz ashraf', 'nawaz sharif', 'shahid khaqan']
```

```
print(len(all_pms))
```

```
7
```

list comprehension

One of the reasons, python is beautiful is because of its poetry like syntax. Consider following loop

```
numbers = []
for i in range(10):
    numbers.append(i**2)
print(numbers)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Now, what if I told you that we can acheive this in one line?

```
numbers = [i**2 for i in range(10)]
print(numbers)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The method of writing for loop inside the list as we have done above is called list comprehension.

We can also have `if` and `else` statement in list comprehension

```
M = [name for name in NH_receivers if 'Muhammad' in name]
print(M)
```

```
['Muhammad Sarwar', 'Tufail Muhammad', 'Muhammad Akran', 'Muhammad Husain Janjua',
↪'Muhammad Mahfuz']
```

```
my_list = [i**2 if i>5 else i**3 for i in range(10)]
print(my_list)
```

```
[0, 1, 8, 27, 64, 125, 36, 49, 64, 81]
```

list comprehension for nested for loops

continue

The `continue` keyword is used inside the for loop when we want to skip some commands in a particular iteration.

```
for president in ['clinton', 'bush', 'obama', 'trump', 'biden']:
    if president == 'trump':
        continue
    # ok sorry they actually wanted to export freedom!
    print(f"{president}: Let's go to war")
```

```
clinton: Let's go to war
bush: Let's go to war
obama: Let's go to war
biden: Let's go to war
```

Above we did not want to print "Let's go to war" when the value of *president* was equal to *trump* so we used `continue` statement.

That was a too simple example. We had better avoid writing *trump* in the list instead of adding two lines inside the for loop. Usually, the conditioning variable (*president* in our case above) appears after doing some calculations inside for loop.

Question: Print all the prime numbers between 1 and 100 using for loop.

break

The `break` keyword is used if we want to stop the iterations of for loop.

```
saving = 5000
for items in ['fridge', 'laptop', 'mobile', 'tablet', 'tv', 'fryer']:
    saving = saving - 1000
    if saving < 0:
        print("no more purchase please!")
        break
```

```
no more purchase please!
```

If you want to dig deep into how the `for` loops work in python, you can jump to *3.16 magic methods*.

Question:

What will be the output of following code?

```
for i in range(0, 3):
    print(a)
    for j in range(0, 2):
        print(j)
        for k in range(0, 1):
            print(k)
```

Total running time of the script: (0 minutes 0.016 seconds)

1.11 print

Important: This lesson is still under development.

`print` is a function in python 3. In general it will print any value given to it as inside argument. If we use two print statements, the next statement is printed on next line.

Print function converts input to string (if it is not already a string) and adds a space at its start if it is not start of newline and puts new line at its end.

```
name = " Ali"
print(1)
print('printing' + name)
```

```
1
printing Ali
```

similar thing happens withing loops. When we put a print function in loop, each print statement is printed on next line.

```
for i in range(5):
    print('printing', i)
```

```
printing 0
printing 1
printing 2
printing 3
printing 4
```

but what if we don't want to print at next line. We can do this by making use of *end* keyword in print function.

```
for i in range(5):
    print(i, end=', ')
```

```
0, 1, 2, 3, 4,
```

The default value of argument `end` is `end=\n`. The print function can take more than one argument to print, and we can separate these arguments by a custom separator using the argument `sep`.

```
print('Iqbal was born in', 1877, sep=': ')
```

```
Iqbal was born in: 1877
```

```
# format
# We can format the output inside print statement by either ``%`` or ``format``. While `%`
# can work both in python 2 and python 3, however, here only `format` is discussed.
```

```
print('Allama Iqbal was born in {} in year {}'.format('Sialkot', 1877))
```

```
Allama Iqbal was born in Sialkot in year 1877
```

```
rivers = {
    "Indus ": [3180000.525, 'Meters'],
    "Chenab ": [760000.850001004, "Meters"],
    "Jhelum": [132.00001245, "KiloMeters"]
}

for river, paras in rivers.items():
    print('River {} is {}  {} long'.format(river, paras[0], paras[1]))
```

```
River Indus  is 3180000.525  Meters long
River Chenab  is 760000.850001004  Meters long
River Jhelum is 132.00001245  KiloMeters long
```

The number of {} must be <= number of arguments in `format`. The ordering of arguments in `format` can be customized as shown below.

```
rivers = {
    "Indus ": [3180000.525, 'Meters'],
    "Chenab ": [760000.850001004, "Meters"],
    "Jhelum": [132.00001245, "KiloMeters"]
```

(continues on next page)

(continued from previous page)

```
}

for river, paras in rivers.items():
    print('River {1} is {0}  {2} long'.format(paras[0], river, paras[1]))
```

```
River Indus  is 3180000.525  Meters long
River Chenab  is 760000.850001004  Meters long
River Jhelum is 132.00001245  KiloMeters long
```

By default {} gets as much space as required by it, but we can fix the space used by a particular {}. The number of spaces must be defined after : inside {}.

```
for river, paras in rivers.items():
    print('River {0:20} is {1:15}  {2:15} long'.format(river, paras[0], paras[1]))
```

```
River Indus                is     3180000.525  Meters          long
River Chenab               is 760000.850001004  Meters          long
River Jhelum               is    132.00001245  KiloMeters      long
```

To define the format of the incoming argument in {}. For example we can use *f* for fractional numbers (as done below). If we don't want fractional part, we can use *d*. For right alignment, we can use <.

```
for river, paras in rivers.items():
    print('River {0:<20} is {1:<15.2f}  {2:<15} long'.format(river, paras[0], paras[1]))
```

```
River Indus                is 3180000.52       Meters          long
River Chenab               is 760000.85        Meters          long
River Jhelum               is 132.00           KiloMeters      long
```

To left align we can use >. e can be used to show numbers in scientific notation.

```
for river, paras in rivers.items():
    print('River {:>20} is {:>15.3e}  {:>15} long'.format(river, paras[0], paras[1]))
```

```
River                Indus  is       3.180e+06           Meters long
River               Chenab  is       7.600e+05           Meters long
River                Jhelum is       1.320e+02       KiloMeters long
```

We probably would have wished to print it like following

```
for river, paras in rivers.items():
    print('River {:<10} is {:^15.1f}  {:<10} long'.format(river, paras[0], paras[1]))
```

```
River Indus      is    3180000.5     Meters     long
River Chenab     is    760000.9      Meters     long
River Jhelum     is      132.0       KiloMeters long
```

^ is used for center alignment. We can truncate long strings as following. If we don't truncate and if incoming string in {} is larger than the space specified, then additional space will be assigned to that {}.

```
print('{:.12}'.format('Khayber Pakhtun Khwa'))
```

```
Khayber Pakh
```

printing list

```
l = [8.364, 0.37, 0.09303, 7.084999, 9.46999999, 0.28600003,
     0.229, 1e-06, 9.414, 0.986001, 2.153005]

print(l)
```

```
[8.364, 0.37, 0.09303, 7.084999, 9.46999999, 0.28600003, 0.229, 1e-06, 9.414, 0.986001,␣
↪2.153005]
```

```
print(*l, sep=', ')
```

```
8.364, 0.37, 0.09303, 7.084999, 9.46999999, 0.28600003, 0.229, 1e-06, 9.414, 0.986001, 2.
↪153005
```

* unpacks the list *l*

```
for p in l: print("{:8.5f}".format(p), end=', ')
```

```
8.36400,  0.37000,  0.09303,  7.08500,  9.47000,  0.28600,  0.22900,  0.00000,  9.41400,␣
↪ 0.98600,  2.15300,
```

dynamic printing

```
import time

for i in range(10):
    print('.', end='')
    time.sleep(0.3)  # This is just to stop the python for 0.3 seconds.
```

```
..........
```

\r removes/deletes what is present on its left side.

```
print('Salam \r alaikum')
```

```
Salam
 alaikum
```

This can be used for more dynamic printing as following

```
a = 0
for x in range (0,10):
    a = a + 1
    b = ("Loading" + "." * a)
    # `b` is printed on top of the previous line.
```

(continues on next page)

(continued from previous page)

```
     # sys.stdout is also called when we use print function
    #sys.stdout.write('\r'+b)
    print('\r'+b, end='')
    time.sleep(0.3)
print ('complete')
```

```
Loading.
Loading..
Loading...
Loading....
Loading.....
Loading......
Loading.......
Loading........
Loading.........
Loading..........complete
```

Following parts of this tutorial are not really necessary and are hardly used but they are mentioned just for the sake of completion.

Customizing print function

We can also add additional features to print function if we wish, by redefining print function, though it will hardly be required.

```
import builtins as _builtins

def MyPrint(*args, **kwargs):
    """My custom print() function."""
    # Adding new arguments to the print function signature
    # is probably a bad idea.
    # Instead consider testing if custom argument keywords
    # are present in kwargs
    _builtins.print('Customized print function')
    return _builtins.print(*args, **kwargs)
```

```
MyPrint('Salam')
```

```
Customized print function
Salam
```

Colored printing

```
print('Normal' + '\033[91m' + ' Red' + '\033[93m' + 'yellow' + '\033[94m' + ' Blue')
```

```
Normal Redyellow Blue
```

```
COLOR = {
    'Bold'       : "\033[1m",
    'Underlined' : "\033[4m",

    'ResetBold'       : "\033[21m",
    'ResetUnderlined' : "\033[24m",

    'Default'      : "\033[39m",
    'Black'        : "\033[30m",
    'Red'          : "\033[31m",
    'Green'        : "\033[32m",
    'Yellow'       : "\033[33m",
    'Blue'         : "\033[34m",
    'Magenta'      : "\033[35m",
    'Cyan'         : "\033[36m",
    'LightGray'    : "\033[37m",
    'DarkGray'     : "\033[90m",
    'LightRed'     : "\033[91m",
    'LightGreen'   : "\033[92m",
    'LightYellow'  : "\033[93m",
    'LightBlue'    : "\033[94m",
    'LightMagenta' : "\033[95m",
    'White'        : "\033[97m",

    'BackgroundDefault'      : "\033[49m",
    'BackgroundBlack'        : "\033[40m",
    'BackgroundRed'          : "\033[41m",
    'BackgroundGreen'        : "\033[42m",
    'BackgroundYellow'       : "\033[43m",
    'BackgroundBlue'         : "\033[44m",
    'BackgroundMagenta'      : "\033[45m",
    'BackgroundCyan'         : "\033[46m",
    'BackgroundLightGray'    : "\033[47m",
    'BackgroundDarkGray'     : "\033[100m",
    'BackgroundLightRed'     : "\033[101m",
    'BackgroundLightGreen'   : "\033[102m",
    'BackgroundLightYellow'  : "\033[103m",
    'BackgroundLightBlue'    : "\033[104m",
    'BackgroundLightMagenta' : "\033[105m",
    'BackgroundLightCyan'    : "\033[106m",
    'BackgroundWhite'        : "\033[107m",
    }

last_color = '\033[91m'
for color, cc in COLOR.items():
    print(last_color + cc)
```

(continues on next page)

(continued from previous page)

```
    last_color = color
```

```
Bold
Underlined
ResetBold
ResetUnderlined
Default
Black
Red
Green
Yellow
Blue
Magenta
Cyan
LightGray
DarkGray
LightRed
LightGreen
LightYellow
LightBlue
LightMagenta
White
BackgroundDefault
BackgroundBlack
BackgroundRed
BackgroundGreen
BackgroundYellow
BackgroundBlue
BackgroundMagenta
BackgroundCyan
BackgroundLightGray
BackgroundDarkGray
BackgroundLightRed
BackgroundLightGreen
BackgroundLightYellow
BackgroundLightBlue
BackgroundLightMagenta
BackgroundLightCyan
```

```
print('ali')
```

```
ali
```

printing special characters

```
print("\N{COPYRIGHT SIGN}")
```

```
©
```

A complete code for special characters can be found here[1]

[1] http://www.fileformat.info/info/charset/UTF-16/list.htm

```
special_characters = {
"NULL": "\N{NULL}",
"SECTION SIGN": "\N{SECTION SIGN}",
"COPYRIGHT SIGN": "\N{COPYRIGHT SIGN}",
"REGISTERED SIGN": "\N{REGISTERED SIGN}",
"INVERTED QUESTION MARK": "\N{LATIN CAPITAL LETTER O WITH STROKE}",
"LATIN CAPITAL LETTER O WITH STROKE": "\N{LATIN CAPITAL LETTER O WITH STROKE}",
"LATIN CAPITAL LETTER SCHWA": "\N{LATIN CAPITAL LETTER SCHWA}",
"LATIN CAPITAL LETTER OPEN E": "\N{LATIN CAPITAL LETTER OPEN E}",
"LATIN SMALL LETTER N WITH LONG RIGHT LEG": "\N{LATIN SMALL LETTER N WITH LONG RIGHT LEG}
↪",
"LATIN CAPITAL LETTER O WITH MIDDLE TILDE": "\N{LATIN CAPITAL LETTER O WITH MIDDLE TILDE}
↪",
"LATIN CAPITAL LETTER ESH": "\N{LATIN CAPITAL LETTER ESH}",
"LATIN SMALL LETTER T WITH HOOK": "\N{LATIN SMALL LETTER T WITH HOOK}",
"LATIN CAPITAL LETTER T WITH RETROFLEX HOOK": "\N{LATIN CAPITAL LETTER T WITH RETROFLEX␣
↪HOOK}",
"LATIN CAPITAL LETTER UPSILON": "\N{LATIN CAPITAL LETTER UPSILON}",
"LATIN SMALL LETTER SCHWA": "\N{LATIN SMALL LETTER SCHWA}",
"LATIN SMALL LETTER ESH": "\N{LATIN SMALL LETTER ESH}",
"LATIN LETTER BILABIAL CLICK": "\N{LATIN LETTER BILABIAL CLICK}",
"GREEK CAPITAL LETTER OMEGA WITH TONOS": "\N{GREEK CAPITAL LETTER OMEGA WITH TONOS}",
"GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS": "\N{GREEK SMALL LETTER IOTA WITH␣
↪DIALYTIKA AND TONOS}",
"GREEK CAPITAL LETTER DELTA": "\N{GREEK CAPITAL LETTER DELTA}",
"GREEK CAPITAL LETTER THETA": "\N{GREEK CAPITAL LETTER THETA}",
"GREEK CAPITAL LETTER LAMDA": "\N{GREEK CAPITAL LETTER LAMDA}",
"GREEK CAPITAL LETTER PI": "\N{GREEK CAPITAL LETTER PI}",
"GREEK CAPITAL LETTER SIGMA": "\N{GREEK CAPITAL LETTER SIGMA}",
"GREEK CAPITAL LETTER PHI": "\N{GREEK CAPITAL LETTER PHI}",
"GREEK CAPITAL LETTER PSI": "\N{GREEK CAPITAL LETTER PSI}",
"GREEK CAPITAL LETTER OMEGA": "\N{GREEK CAPITAL LETTER OMEGA}",
"GREEK SMALL LETTER ETA WITH TONOS": "\N{GREEK SMALL LETTER ETA WITH TONOS}",
"GREEK SMALL LETTER IOTA WITH TONOS": "\N{GREEK SMALL LETTER IOTA WITH TONOS}",
"GREEK SMALL LETTER UPSILON WITH DIALYTIKA AND TONOS": "\N{GREEK SMALL LETTER UPSILON␣
↪WITH DIALYTIKA AND TONOS}",
"GREEK SMALL LETTER ALPHA": "\N{GREEK SMALL LETTER ALPHA}",
"GREEK SMALL LETTER BETA": "\N{GREEK SMALL LETTER BETA}",
"GREEK SMALL LETTER GAMMA": "\N{GREEK SMALL LETTER GAMMA}",
"GREEK SMALL LETTER DELTA": "\N{GREEK SMALL LETTER DELTA}",
"GREEK SMALL LETTER EPSILON": "\N{GREEK SMALL LETTER EPSILON}",
"GREEK SMALL LETTER ZETA": "\N{GREEK SMALL LETTER ZETA}",
"GREEK SMALL LETTER ETA": "\N{GREEK SMALL LETTER ETA}",
"GREEK SMALL LETTER THETA": "\N{GREEK SMALL LETTER THETA}",
"GREEK SMALL LETTER LAMDA": "\N{GREEK SMALL LETTER LAMDA}",
"GREEK SMALL LETTER MU": "\N{GREEK SMALL LETTER MU}",
"GREEK SMALL LETTER NU": "\N{GREEK SMALL LETTER NU}",
"GREEK SMALL LETTER XI": "\N{GREEK SMALL LETTER XI}",
"GREEK SMALL LETTER PI": "\N{GREEK SMALL LETTER PI}",
"GREEK SMALL LETTER SIGMA": "\N{GREEK SMALL LETTER SIGMA}",
"GREEK SMALL LETTER TAU": "\N{GREEK SMALL LETTER TAU}",
"GREEK SMALL LETTER CHI": "\N{GREEK SMALL LETTER CHI}",
"GREEK SMALL LETTER PSI": "\N{GREEK SMALL LETTER PSI}",
```

(continues on next page)

(continued from previous page)

```
"GREEK SMALL LETTER OMEGA": "\N{GREEK SMALL LETTER OMEGA}",
"GREEK PHI SYMBOL": "\N{GREEK PHI SYMBOL}",
"GREEK SMALL LETTER ARCHAIC KOPPA": "\N{GREEK SMALL LETTER ARCHAIC KOPPA}",
"GREEK CAPITAL THETA SYMBOL": "\N{GREEK CAPITAL THETA SYMBOL}",
"GREEK LUNATE EPSILON SYMBOL" : "\N{GREEK LUNATE EPSILON SYMBOL}"
}

for sp, val in special_characters.items():
    print("{:52} {:>5}".format(sp, val))
```

```
NULL                                                SECTION SIGN                ␣
↪                          §
COPYRIGHT SIGN                                      ©
REGISTERED SIGN                                     ®
INVERTED QUESTION MARK                              Ø
LATIN CAPITAL LETTER O WITH STROKE                  Ø
LATIN CAPITAL LETTER SCHWA
LATIN CAPITAL LETTER OPEN E
LATIN SMALL LETTER N WITH LONG RIGHT LEG
LATIN CAPITAL LETTER O WITH MIDDLE TILDE
LATIN CAPITAL LETTER ESH
LATIN SMALL LETTER T WITH HOOK
LATIN CAPITAL LETTER T WITH RETROFLEX HOOK
LATIN CAPITAL LETTER UPSILON
LATIN SMALL LETTER SCHWA
LATIN SMALL LETTER ESH
LATIN LETTER BILABIAL CLICK
GREEK CAPITAL LETTER OMEGA WITH TONOS
GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS
GREEK CAPITAL LETTER DELTA
GREEK CAPITAL LETTER THETA
GREEK CAPITAL LETTER LAMDA
GREEK CAPITAL LETTER PI
GREEK CAPITAL LETTER SIGMA
GREEK CAPITAL LETTER PHI
GREEK CAPITAL LETTER PSI
GREEK CAPITAL LETTER OMEGA
GREEK SMALL LETTER ETA WITH TONOS
GREEK SMALL LETTER IOTA WITH TONOS
GREEK SMALL LETTER UPSILON WITH DIALYTIKA AND TONOS
GREEK SMALL LETTER ALPHA
GREEK SMALL LETTER BETA
GREEK SMALL LETTER GAMMA
GREEK SMALL LETTER DELTA
GREEK SMALL LETTER EPSILON
GREEK SMALL LETTER ZETA
GREEK SMALL LETTER ETA
GREEK SMALL LETTER THETA
GREEK SMALL LETTER LAMDA
GREEK SMALL LETTER MU
GREEK SMALL LETTER NU
GREEK SMALL LETTER XI
GREEK SMALL LETTER PI
```

```
GREEK SMALL LETTER SIGMA
GREEK SMALL LETTER TAU
GREEK SMALL LETTER CHI
GREEK SMALL LETTER PSI
GREEK SMALL LETTER OMEGA
GREEK PHI SYMBOL
GREEK SMALL LETTER ARCHAIC KOPPA
GREEK CAPITAL THETA SYMBOL
GREEK LUNATE EPSILON SYMBOL
```

Total running time of the script: (0 minutes 6.021 seconds)

1.12 functions

Important: This lesson is still under development.

A function is a box which takes something as input and it gives you output after performing some operations on that input. In python, the input and output arguments for a function are optional. The basic syntax of a minimal function in is as below

```
def FunctionName(InputArguments):
    commands to find output
    return output
```

As said earlier, *InputArguments* and `return` are optional. This means we can write a valid function in python without input arguments or a function which does not return something.

```
def add_nums():
    pass

print(add_nums)
```

```
<function add_nums at 0x7f136952b160>
```

```
type(add_nums)
```

If we want to use/run a function, this means we want to **call** it. Following syntax can be used to call a function

```
Output = FunctionName(InputArguments)
```

`Output` is optional and so does `InputArguments`

```
add_nums()
```

The following function takes two input arguments and adds them

```
def add_nums(a, b):  # for brevity, we can not write add_nums(a+b):
    print('a: ', a, ' b:', b)
    a + b

add_nums(1, 5)
```

```
a:  1  b: 5
```

Inside the function, we name the two input arguments as *a* and *b*.

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    a + b

x = 12
y = 14
add_nums(x, y)
```

```
a:  12  b: 14
```

It is important to understant that the variable *a* and *b* are created once we are inside the function `add_nums`.

Outside function *12* is *x* but inside function, *12* is *a*. The variables *a* and *b* are not available outside the function.

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b
    return c

x = 12
y = 14
add_nums(x, y)
```

```
a:  12  b: 14

26
```

Above we we created *c* inside the function and returned it using the `return` statement. Outside the function when we *called* the function *add_nums* by executing `add_nums(x,y)`, this value of *c* is printed. The variable named *c* itself is not available outside the function. The variables which can be seen inside the function and which can be seen outside the function will be covered in *1.14 global vs local*.

Return

A function returns `None` by default. The value returned by a function can be assigned to a new variable for example to *x* in following example. It can be any legal variable name though.

```
import random  # ignore this line if you don't know what it does
qatleen = ['winsten churchil', 'rana sanaullah', 'obama', 'musharaf']

def print_qatal():
    print(random.choice(qatleen))
    return None

x = print_qatal()
print('type of x: ', type(x))
```

```
obama
type of x:  <class 'NoneType'>
```

Even if a function does not return anything explicitly, it still returns `None`.

```
def print_qatal():
    print(random.choice(qatleen))
    return


x = print_qatal()
print('type of x: ', type(x))
```

```
rana sanaullah
type of x:  <class 'NoneType'>
```

If a function does not have `return` statement, it still returns `None`.

```
def print_qatal():
    print(random.choice(qatleen))

x = print_qatal()
print('type of x: ', type(x))
```

```
obama
type of x:  <class 'NoneType'>
```

So it is impossible in python to write a function which returns absolutely nothing.

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b
    return c


x = 12
y = 14
z = add_nums(x, y)
print('The function returns z: ', z)
```

```
a:  12  b: 14
The function returns z:  26
```

The variables c and z have same values with the difference that c exists only inside the function. Moreover, the creation of c is not necessary, we can just return the result as it is.

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    return a + b  # no intermediate variable c

x = 12
y = 14
```

(continues on next page)

(continued from previous page)

```
z = add_nums(x, y)
print('The function returns z: ', z)
```

```
a:  12  b: 14
The function returns z:  26
```

Question: What value will be printed as a result of code in following cell?

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    z = a + b

x = 12
y = 14
z = add_nums(x, y)
print('The function returns z: ', z)
```

Question: What value will be printed as a result of code in following cell?

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b

x = 12
y = 14
add_nums(x, y)
print('The value of c is: ', c)
```

Question: What value will be printed as a result of code in following cell?

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b
    return c

x = 12
y = 14
add_nums(x, y)
print('The value of c is: ', c)
```

Question: What value will be printed as a result of code in following cell?

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b
    return c

x = 12
y = 14
z = add_nums(x, y)
print('The value of c is: ', c)
```

Question: What value will be printed as a result of code in following cell?

```
def add_nums(a, b):
    print('a: ', a, ' b:', b)
    c = a + b
    return c

x = 12
y = 14
z = add_nums(x, y)
print('The value of z is: ', c)
```

Default values of input arguments

We can provide default values to input arguments. The default values of input arguments are only used if we don't provide their values when calling them, otherwise their values are overwritten.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

x = 114
y = 313
z = add_nums(x, y)
print('The function returns z: ', z)
```

```
a:  114  b: 313
The function returns z:  427
```

Above: The value from variable *x* is going to *a* and will replace its default value. The value from variable *y* will go to *b* and will overwrite its default value i.e. 14.

When we have defined the default values of input arguments in function definition, we can skip one or more input arguments when calling the function as shown below.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

y = 313
z = add_nums(y)
print('The function returns z: ', z)
```

```
a:  313  b: 14
The function returns z:  327
```

Above, the value from *y* will be assigned to *a* while *b* will use its default value.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

y = 313
```

(continues on next page)

(continued from previous page)

```
z = add_nums(b=y)
print('The function returns z: ', z)
```

```
a:  12  b: 313
The function returns z:  325
```

Above, *a* will use its default value while *b* will get value of *y*.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

x = 114
z = add_nums(a=x)
print('The function returns z: ', z)
```

```
a:  114  b: 14
The function returns z:  128
```

Above: *a* will use the value of *x* i.e. 114 while *b* will use its default value i.e. 14.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

z = add_nums(1)
print('The function returns z: ', z)
```

```
a:  1  b: 14
The function returns z:  15
```

Above, *1* will go to *a* while *b* will use its default value. The function can also be called without providing any input argument because both input arguments are optional. In this case the default values will be used.

```
def add_nums(a=12, b=14):
    print('a: ', a, ' b:', b)
    return a + b

z = add_nums()
print('The function returns z: ', z)
```

```
a:  12  b: 14
The function returns z:  26
```

We can define the optional arguments with obligatory arguments. In function below, *c* is optional, while *a* and *b* are obligatory.

```
def add_nums(a, b, c=14):
    print('a:', a, '  b:', b, '  c:', c)
    return a + b
```

(continues on next page)

(continued from previous page)

```
z = add_nums(1, 2)
print('The function returns z: ', z)
```

```
a: 1   b: 2   c: 14
The function returns z:  3
```

Above, we have not provided the value for *_c*. As it was optional argument, its default value was used.

Question: What will be the output of following function?

```
z = add_nums()
print('The function returns z: ', z)
```

Question: Guess the output from following cell.

```
z = add_nums(a=1, c=313)
 print('The function returns z: ', z)
```

Returning multiple values

If a function returns more than one object/variables, and we assign it to a single variable, then this new variable will be `tuple`.

```
def func(a, b):
    u = a
    v = b
    return u, v

xx = func(5, 12)
print(xx, type(xx))
```

```
(5, 12) <class 'tuple'>
```

Above: The function returns 2 arguments but we assigned it to 1 variable named *xx*. Thus, *xx* will be a tuple consisting of two values

We can however, assign both returned values from a function to two new variables as shown below. %%

```
xx, yy = func(5, 12)
print(xx, type(xx))
print(yy, type(yy))
```

```
5 <class 'int'>
12 <class 'int'>
```

If the function returns fewer or more arguments than the variables assigned, then it will give error.

```
# uncomment the following line
# xx, yy, zz = func(5, 12)  # Error
```

If the function returns multiple values but we want to get only first value, we can do it as below

```
xx = func(15, 12)[1]
print(xx)
```

```
12
```

Above, we are slicing the output of *func* using *[1]* to get the second value. The type of the returned value will be the same as the type of the value returned by the function.

```
def retun_list(a):
    return [a]

out = retun_list(10)
print(type(out))
```

```
<class 'list'>
```

A function can return a tuple in following ways

```
def return_tuple(variable):
    return variable, variable

out = return_tuple(10)
print(type(out))
```

```
<class 'tuple'>
```

```
def return_tuple(variable):
    return variable,

out = return_tuple(10)
print(type(out))
```

```
<class 'tuple'>
```

```
def return_tuple(variable):
    return (variable)

out = return_tuple(10)
print(type(out))
```

```
<class 'int'>
```

The type of *out* is `int` because it is comma `,` which makes something a tuple not the brackets. So in order to get a tuple from function, we must put comma even if there is only one object in the tuple as shown below.

```
def return_tuple(variable):
    return (variable, )

out = return_tuple(10)
type(out)
```

```
def return_tuple(variable):
    return variable, variable+2, variable+4

var, *junk = return_tuple(2)

print(type(var), var)
print(type(junk), junk)
```

```
<class 'int'> 2
<class 'list'> [4, 6]
```

A common way to ignore the unncessary output from a function is to use underscore

```
var, *_ = return_tuple(2)
```

Question: What will be the length of _ above?

Above we were interested in only *var* and wanted to ignore everything else returned by *return_tuple* function.

calling a function from another function

```
def foo(a):
    return a + 1

def bar(a):
    return foo(a) + 1

bar(1)
```

```
3
```

Above we are calling function *foo* from inside function *bar*. The output of *foo* is added with 1 and returned by *bar*.

```
def foo(a):
    return a + 1

def bar(a):
    return a+100

def baz(a):
    b = foo(a)
    b = bar(b)
    return b

baz(1)
```

```
102
```

Above we are calling function *foo* from inside function *baz*. The output of *foo* is assigned to variable *b* and then *bar* is called with *b* as input argument. The output of *bar* is again assigned to *b* and returned by *baz*.

Question:

```
def add_nums(a, b):
    return a + b

def sub_nums(a, b):
    return a - b

def multiply_nums(a, b):
    return a * b

def divide_nums(a, b):
    return a / b
```

Write a function named *foo* in such a way that you call all the above four functions (*add_nums*, *sub_nums*, *multiply_nums*, *divide_nums*) from inside *foo*. The *foo* function should take two input arguments and return 100. You can create intermediate variables as well.

function as input argument

The input arguments to a function can be any python object. This includes functions as well. We can assign a function to a variable and use that variable to call the function.

```
def print_me(to_print):
    print(to_print)

x = print_me

x('This goes into print_me')
```

```
This goes into print_me
```

Thus we can use functions as input arguments to other functions as well.

```
def magic(left, op, right):
    return op(left, right)

def my_op(var_a, var_b):
    return var_a == var_b

magic(2, my_op, 2)
```

```
True
```

positioning of `return` statement

It is important to understand that the function will exit as soon as it encounters a `return` statement. This means that the code after the `return` statement will not be executed.

```
def add_nums(a, b):
    return a + b
    print('This will not be printed')

add_nums(1, 2)
```

```
3
```

```
def add_nums(a_list, break_point=5):
    _sum = 0.0
    for val in a_list:
        _sum += val
        if _sum > break_point:
            break
    return _sum


x = add_nums([1, 2, 3, 4, 5])
print(x)
```

```
6.0
```

As soon as the value of *_sum* became greater than *break_point*, the `for` loop exited and we got the value of *_sum* at that point.

```
x = add_nums([1, 2, 3, 4, 5], 50)
print(x)
```

```
15.0
```

Question: Was the `break` statement executed above?

Question: What will be the output of following code?

```
def foo(a,b):
    c = a+b
    return c
    c = a-b
z = foo(1,2)
print(z)
```

Question: What will be the output of following code?

```
def foo(a,b):
    c = a+b
    return c
    c = a-b
    return c
```

(continues on next page)

(continued from previous page)

```
z = foo(1,2)
print(z)
```

Question: What will be the output of following code?

```
def foo(a,b):
    if a==0:
        return b
    else:
        return foo(a-1, a+b)
```

docstring

The first string inside the functions is usually put for help. This is called *docstring*. It can be called by __doc__ method

```
def fahrenheit(T_in_celsius):
    """This function converts temperature from Celsius to Fahrenheit. """
    return (T_in_celsius * 9 / 5) + 32

fahrenheit.__doc__
```

```
'This function converts temperature from Celsius to Fahrenheit. '
```

```
help(fahrenheit)
```

```
Help on function fahrenheit in module __main__:

fahrenheit(T_in_celsius)
    This function converts temperature from Celsius to Fahrenheit.
```

If we want to know the name of a function as string we can do it as following.

```
converter = fahrenheit

print(converter.__name__)
```

```
fahrenheit
```

Total running time of the script: (0 minutes 0.016 seconds)

1.13 args and kwargs

Important: This lesson is still under development.

*args

We have learned about functions that they can take one or more input arguments. If we want our function to have variable number of input arguments, one way to do this is to put asterik * before the *InputArgument* name inside () during function definition. The common practice is to use `args` as *InputArgument* name in this case. Thus it becomes `*args`. This allows us to have multiple **unnamed** input arguments.

```
def add_nums(*args):
    print(type(args), len(args), args)


add_nums(5, 12.0)
```

```
<class 'tuple'> 2 (5, 12.0)
```

Above `*args` is essentially converting all the unnamed input arguments to the function *add_nums* into tuple. Inside the above function, the unnamed input arguments 5 and 12 become *(5, 12)* tuple.

We can also say that `*args` is packing all the unnamed input arguments into a tuple.

```
l = [2, 3, 4]
add_nums(l)
```

```
<class 'tuple'> 1 ([2, 3, 4],)
```

Above we provided a single unnamed input argument *l* to the function *add_nums*. THerefore the length of *args* is 1. Inside the function *add_nums*, the input argument *[2,3,4]* which was a list, is taken as *([2,3,4],)* which is tuple.

Since `*args` is converting all the unnamed input arguments to tuple, which is a sequence, so we can iterate over it as shown below.

```
def do_add(*args):
    print(len(args), 'is length of args in do_add')
    for arg in args:
        print(arg)
    return

def add_nums(*args):
    # we get a tuple i.e. (12,14)
    print(len(args), 'is length of args in add_nums')

    # we pass the tuple (12,14) as input argument and NOT 12,14 as two input arguments
    return do_add(args)


add_nums(12, 14)
```

```
2 is length of args in add_nums
1 is length of args in do_add
(12, 14)
```

There was just one iteration in the `for` in *do_add* function above. This is because the input argument to *do_add* was a tuple *(12,14)* and not two input arguments *12, 14*.

We have seen that *args is used to pack all the unnamed input arguments into a tuple. It can also unpack a tuple/list into individual unnamed input arguments as shown below.

```
def do_add(*args):
    print(len(args), 'in do_add')
    return

inputs = [1,2,3]

do_add(*inputs)
```

```
3 in do_add
```

Above, the list *inputs* is unpacked into individual input arguments *1*, *2* and *3*.

```
def do_add(*args): # packing all unnamed input arguments into a tuple
    print(len(args), 'in do_add')
    a = 0
    for arg in args:
        a += arg
    return a


def add_nums(*args):  # packing all unnamed input arguments into a tuple
    print(len(args), 'in add_nums')
    # we get a tuple i.e. (12,14) as input

    # we are again providing input as 2,3 which means we are providing two inputs
    return do_add(*args)  # unpacking tuple into individual input arguments


add_nums(12, 14)
```

```
2 in add_nums
2 in do_add

26
```

Above: *add_nums* and *do_add* are called with exactly same kind of input arguments.

Note that how the packing and unpacking is being done in the above code using *args.

Following is an example of misplaced return statement

```
def do_add(*args):
    print(len(args), 'in do_add')
    a = 0
    for arg in args:
        a += arg
        return a


def add_nums(_a, *args):
    print(len(args), "in add_nums")
```

(continues on next page)

(continued from previous page)

```
    return do_add(_a, *args)  # this asterik unpacks tuple args


add_nums(2, 12, 14)
```

```
2 in add_nums
3 in do_add

2
```

Above, *args is packing all the unnamed input arguments (12 and 14) into a tuple. After that by the statement `do_add(_a, *args)` the tuple (12,14) is unpacked into individual input arguments by the asterik. The function *do_add* is called with exactly 3 input arguments. In the function *do_add*, the asterik is used to pack all the three input arguments into a tuple whose length is three. The *return* statement in *do_add* function is misplaced in the function *do_add* which is causing the function to return after first iteration of the loop.

```
def do_add(a, *args):
    print(len(args), 'in do_add')

    for arg in args:
        a += arg
    return a


def add_nums(a, *args):
    print(len(args), "in add_nums")
    return do_add(a, *args)


add_nums(5, 12, 14)
```

```
2 in add_nums
2 in do_add

31
```

In the above cell *a* is the positional argument which takes 5, while while 12 and 14 are given as tuple to the function

```
def add_nums(a, b, c):
    print(a, b, c)
    return


l = [1, 5, 12]

add_nums(*l)
```

```
1 5 12
```

In above code, the list *l* is unpacked into individual input arguments *1*, *5* and *12*. The function *add_nums* is called with exactly 3 input arguments.

if the list *l* contains more than 3 elements, passing it will raise error because in this case the number of input arguments

will be more than 3.

```
# uncomment following line
# l = [1, 5, 12, 3]
# add_nums(*l)
```

Question:

What will be printed when we execute the following code!

```
def foo(a, *b):
    print(len(b))

inputs = [1,2,3,4]
foo(*inputs)
```

Although we can vary the number of arguments by using **args* but we can not know the name of input arguments.

**kwargs

We saw that `*args` is used to pack all the unnamed input arguments into a tuple. It can also unpack a tuple/list into individual unnamed input arguments. If we want our function to have variable number of **named** input arguments, we can achieve this by putting double asterik `**` before the *InputArgument* name inside `()` during function definition. The common practice is to use `kwargs` as *InputArgument* name in this case. Thus it becomes `**kwargs`. `**kwargs` is used to pack all the named input arguments into a dictionary. Similarly, it can also unpack a dictionary into individual named input arguments.

```
def add_nums(**kwargs):
    print(type(kwargs))


add_nums(a=5, b=12)  # we can not do add_nums(2,3) here
```

```
<class 'dict'>
```

It is just a convention to use the word *kwargs* for keyword arguments. We can put any other name as well. For example, the above function can also be written as below.

```
def add_nums(**dictionary):
    print(type(dictionary), len(dictionary))


add_nums(a=5, b=12)
```

```
<class 'dict'> 2
```

```
def add_nums(**kwargs):
    print(type(kwargs))


a = 5
b = 12
dictionary = {'a': a, 'b': b}
```

If a function has only **kwargs as input argument, we can not provide unnamed input arguments to the function. For example, following code will raise error.

```
# uncomment following line to see the error
#
# add_nums(a,b)
```

uncomment following line to see the error

```
# add_nums(dictionary)
```

add_nums(a,b) is invalid because we are providing two unnamed input arguments. However, *add_nums(dictionary)* ` is invalid because we are providing a single unnamed input argument.

```
add_nums(**dictionary)
```

```
<class 'dict'>
```

```
def add_nums(**kwargs):
    x = kwargs['a']
    y = kwargs['b']
    return x + y


a = 5
b = 12
dictionary = {'a': a, 'b': b}
add_nums(**dictionary)
```

```
17
```

Omitting ** on both sides also serves the purpose. However, if we omit ** in function definition, it would mean that function requires an input argument named `kwargs`. If we ommit ** when calling the function it means that we are giving a dicionary as it is as input argument.

```
def add_nums(kwargs):
    x = kwargs['a']
    y = kwargs['b']
    return x + y


a = 5
b = 12
dictionary = {'a': a, 'b': b}
add_nums(dictionary)
```

```
17
```

```
def add_nums(a=1, b=5):
    print(type(a))
    return a + b
```

(continues on next page)

(continued from previous page)

```
d = {'a': 12, 'b': 14}
add_nums(**d)
```

```
<class 'int'>

26
```

```
def add_nums(xx, **kwargs):
    x = kwargs['a']
    y = kwargs['b']
    return xx + x + y


dictionary = {'a': 5, 'b': 12}
add_nums(1, **dictionary)
```

```
18
```

xx is assigned the value of *1* when the function *add_nums* is called. We can put extra named arguments separately along with ***kwargs*.

```
def add_nums(xx=1, **kwargs):
    _sum = 0.0
    for key, val in kwargs.items():
        _sum += val

    return _sum


dictionary = {'a': 5, 'b': 12}
add_nums(**dictionary)
```

```
17.0
```

The value of *xx* which is *1* is not added in the final answer.

```
def add_nums(xx=1, yy=12, zz=14, **kwargs):
    _sum = 0.0
    for key, val in kwargs.items():
        _sum += val

    return _sum


add_nums(xx=114, yy=313, zz=7, a=1, b=5, c=12, d=14)
```

```
32.0
```

now *kwargs* contained *a*, *b*, *c* and *d* only. The above code can also be written as following.

```
def add_nums(xx=1, yy=12, zz=14, **kwargs):
    _sum = 0.0
    for key, val in kwargs.items():
        _sum += val

    return _sum


additional_args = {
    "a": 1,
    "b": 5,
    "c": 12,
    "d": 14,

}
add_nums(xx=114, yy=313, zz=7, **additional_args)
```

```
32.0
```

A misplaced *return* can be a cause of many headachs, as in following case.

```
def add_nums(xx=1, yy=12, zz=14,
             **kwargs):
    # if number of arguments are large, it is totally fine to go on
    # second line without any probelm

    _sum = 0.0
    for key, val in kwargs.items():
        _sum += val

        return _sum


additional_args = {
    "a": 114,
    "b": 5,
    "c": 12,
    "d": 14,

}
add_nums(xx=114, yy=313, zz=7, **additional_args)
```

```
114.0
```

Above, the function *add_nums* finishs execution just after first iteration of *for loop*.

```
def cook_lunch_with_bread(raw_food):
    print('prepare lunch with break')


def cook_lunch_without_bread(raw_food):
    print('prepare lunch with {}, {} and without bread'.format(*raw_food.keys()))
```

(continues on next page)

(continued from previous page)

```
def lunch(**raw_food):
    if 'bread' in raw_food:
        cook_lunch_with_bread(raw_food)
    else:
        cook_lunch_without_bread(raw_food)


lunch(rice='2', milk=1)
```

```
prepare lunch with rice, milk and without bread
```

We can predefine/fix the number of positional arguments and keyword arguments that a function must take. Following code allows only 3 positional keyword arguments and 2 keyword arguments.

```
def add_nums(a, b, c, *, d, e):
    print(a, b, c)
```

we can not do add_num(2,3,4, 12, 14), as this would mean 5 positional arguments instead of 3.

```
add_nums(2, 3, 4, d=12, e=14)
```

```
2 3 4
```

We can make use of `*args` and `**kwargs` together in a function definition. In this case , our function can receive any number of unnamed and named input arguments.

```
def add_nums(a, b, *args, **kwargs):
    _sum = a + b
    print(len(args), len(kwargs))
    for arg in args:
        _sum += arg
    for key, val in kwargs.items():
        _sum += val

    return _sum


add_nums(2, 3)
```

```
0 0

5
```

```
add_nums(2, 3, 4, 5)
```

```
2 0

14
```

4 and *5* goes to args

```
add_nums(2, 3, 4, 5, 6, d=5, e=12)
```

```
3 2

37
```

4, *5*, *6* goes to args and *d* and *e* goes to kwargs

We can replace the words *args* and *kwargs* with any other variable name. It is just a convention to use the word *args* for unnamed arguments and the word *kwargs* for keyword arguments. The actual packing and unpacking is done by ``*`` and ``**`` in python. For example, the above function can also be written as below

```
def add_nums(one, five, *unnamed, **named):
    _sum = one + five
    print(len(unnamed), len(named))
    for arg in unnamed:
        _sum += arg
    for key, val in named.items():
        _sum += val

    return _sum


add_nums(1, 5, 12, infallibles=14, messangers=313)
```

```
1 2

345
```

Question: What will be the output of following code?

Question: Write a function named *func* which can be called as below and always returns 10. Remember, it should be the same function, but we will call it in five different ways, and it should always return 10.

```
func(1)

func(a=1, b=2)

func(1, b=2)

func(1,2,3,4)

func(a=1, b=2, c=3, d=4)
```

Question: In the above functions, what are the number of `args` and `kwargs` at each of the five calls that we have made above?

Question: Write a function *func* which receives x and y along with `args` and `kwargs` and call this function with 5 input arguments in such a way that it returns 10. Inside the function, print the length of args and kwargs and it must be equal to 1. Remember, the as per above instructions, the signature of *func* must be as below

```
def func(x, y, *args, **kwargs):
    ...
```

Question: Call the above defined function (without modifying the function) so that the length of `args` and `kwargs` is greater than 1. Remember that the function must return 10.

Question:

```
def foo(**suggestionsf):
    pass

foo([1,2]) # -> error
```

Running the above code will raise error. Why?

Total running time of the script: (0 minutes 0.012 seconds)

1.14 global vs local

It is very important to understand the difference between global scope and local scope of a variable/object in python. A variable defined outside a function has global scope while a variable defined inside a function has local scope.

```
var = 1
def foo():
    return
```

Above the variable `var` has global scope. We can access it from anywhere in the script.

```
def foo():
    var1 = 1
    return
```

The variable `var1` has local scope. We can access, use and modify it only insdie the function `foo`. We can not access it from outside the function `foo`. If we try to access it from outside the function, python will give us an error. Hence the variable `var1` has local scope.

If a variable is defined outside the function and a variable with same name is defined and(or) being modified inside a function, then that definition/modification inside the function will have no effect on the variable outside the function.

```
var = 1

def foo(n):
    var = n
    print(var, 'inside foo')
    return

foo(10)

print(var, 'outside foo')
```

```
10 inside foo
1 outside foo
```

var inside the *foo* is different than the *var* outside the *foo*. If we try to modify the value of a variable (with some exceptions) which is defined only outside the function, it will result in error.

```
var = 1
```

(continues on next page)

(continued from previous page)

```
def foo(n):
    print(n)
    var = var + n
    return var
```

uncomment follwoing line

```
# foo(10)  # UnboundLocalError
```

Read and try to make sense of the error message as understanding error messages is one of the best ways to master a programming language. So the variable *var* which is defined outside *foo* can not be accessed (actually it can be accessed but not changed/reassigned, discussion follows) inside *foo*. If we want to do so, we need to use the keyword `global`.

```
var = 1

def foo(n):
    global var
    print(n)
    var = var + n
    return var

foo(10)
```

```
10

11
```

By making use of `global` statement, we are saying that the variable *var* inside the function is same object as the variable *var* outside the function *foo*. Similarly if a variable is defined inside the function, we can not access that variable outside the function.

```
def foo(n):
    var1 = n+2
    return

foo(10)
```

```
# uncomment following line

# print(var1) NameError
```

variable *var1* is defined inside the function and is removed from memory as soon as the function execution finishes at *return* statement. (Why I used *var1* instead of *var* for this example?) If we want to access a variable -which is defined inside the function- outside the function, we have to make this variable global by making use of `global` statement.

```
def foo(n):
    global var1
    var1 = n+2
    return

foo(10)
```

```
# uncomment following line
# print(var1)
```

when python finds a variable is local

```
var = 1

def foo(n):
    print(var)

foo(10)
```

```
1
```

The above code shows we are able to access/employ/use/read the value of variable *var* inside *foo* without making it *global*.

```
var = 1

def foo(n):
    print(var)
    var = 0
```

```
# uncomment following line
# foo(10)  # UnboundLocalError
```

So untill we defined the variable *var* inside *foo* by *var=0*, python did not create the local variable *var*. Until that point if we use/access the value of *var*, python will give us the value of *var* from outer scope. But as soon as we defined *var* inside the function *foo*, then python knows that this *var* is a local variable and is different than *var* outside the *foo*. Python then creates the local variable *var*. At this point python makes the variable *var*, **local**. (You can say, python forgets what *var* in outer scope is). As we tried to use *var* before declaring it global, hence the error message.

```
var = 1

def foo(n):
    global var
    var = n  # modifying global copy of var

def print_var():
    print(var)  # prints global value of var

print(var)
foo(10)
print_var()
```

```
1
10
```

Takeaway: We need to declare a variable as *global* in a function which assigns a value of it. If we want to ONLY USE a global variable in local scope, we can do this without declaring it global. However, this is error prone and should be avoided. For example, following code works but it is not recommended.

```
var = 1
def foo(n):
    return n+var

foo(10)
```

```
11
```

The above code works but it is not recommended. It is better to declare the variable as global if we want to use it in local scope. Or we can pass the variable as argument to the function.

```
var = 1
def foo(n, var):
    return n+var

foo(10, var)
```

```
11
```

The above code is better than the previous one. We are passing the variable *var* as argument to the function *foo*. However, the name of the variable *var* inside the function *foo* is same as the name of the variable *var* outside the function. This is also not recommended since it is error prone and many modern IDEs will raise warnings. We should use different names for variables in different scopes. The following code is better than the previous one.

```
var = 1
def foo(n, var1):
    return n+var1

foo(10, var)
```

```
11
```

alternative to global

If we want to share a variable between two functions, and the outer scope, we have to make use of `global` statement in both functions.

```
var = 0   # The initial value of x, with global scope

def foo2():
    global var
    var = var + 5

def main():
    global var   # So we can change the value of var inside main
    print(var)    # first check the var inside main
    var = 10
    print(var)
    foo2()
    print(var)
```

(continues on next page)

(continued from previous page)

```
main()
```

```
0
10
15
```

We should normaly avoid sharing variables among different functions with the help of `global`. The alternative to do this, is to make use of functions with `return` statement. The above code can be written as follows.

```
def foo1(parameter):
    return parameter + 5

def main(var):
    print(var)
    var = 10
    print(var)
    var = foo1(var)
    print(var)

main(0)
```

```
0
10
15
```

existence of a vriable

We have already created the variable *var* previously. Now the variable *var* exists in memory. We can print its value.

```
print(var)
```

```
15
```

We can delete the variable from memory by making use of *del* statement.

```
del var
```

Now if we try to print the value of *var* again, python says it does not know what is *var*, since we removed it from memory.

```
# uncomment following line
# var  # NameError
```

If we try to delete a variable which does not exist in memory, python will raise an error.

```
# uncomment following line
#
# del var  # NameError
```

If we want to safely remove/delete a variable, we can first check whether it is present in memory or not and delete it only if it is present as shown below.

```
_all = locals().copy()
_all.update(globals())
if 'var' in globals():
    del var
    print('var removed')
else:
    print('it was not in memory')
```

```
it was not in memory
```

We can create the variable *var* once again and try to remove it using the above method to validate it.

```
var = 3
var
```

```
3
```

```
if 'var' in globals():
    del var
    print('var removed')
else:
    print('it was not in memory')
```

```
var removed
```

`globals()` returns a dictionary of all global variables while `locals()` returns a dictionary of all local variables. The names of variables are keys of these dictionaries and values of these variables are values of these keys in these dictionaries.

```
for k,v in locals().items():
    print(k)#, v)  # not printing values for brevity
```

```
for k,v in globals().items():
    print(k)
```

We can call a function by its string name With the help of `globals()` ` function. In this way we can call a method by its `string` name.

```
def qatal(a):
    print(a + ' sanaullah')

func_as_string = 'qatal'

globals()[func_as_string]('rana') # qatal().
```

```
rana sanaullah
```

`globals()[func_as_string]('rana')` is equivalent to `qatal('rana')`.

Normally global variables are considered bad see this[1] and this[2] . In python, by convention, global is used for constants and variables are seldom used as global. Technically in python there is no difference between variables and constants,

[1] http://wiki.c2.com/?GlobalVariablesAreBad
[2] https://en.wikipedia.org/wiki/Side_effect_(computer_science)

however it is a convention to capitalize GLOBAL CONSTANTS and not global_variables. It is recommende that you explicitly declare global inside a function when you are using a global variable even though if it is not required.

If there is a local variable with same name as global variable and we want to modify global variable, we can make use of `globals()`.

```
thug = 'showbaz'

def commision(accused):
    thug = 'nawaz'
    print('internal thug: ', thug)
    globals()['thug'] = accused
    print('internal thug: ', thug)
    return

print('global thug before: ', thug)
commision('bajwa')
print('global thug later:', thug)
```

```
global thug before:  showbaz
internal thug:  nawaz
internal thug:  nawaz
global thug later: bajwa
```

Mutable objects

Above, we saw that if we want to modify the value of a variable which is defined outside the function, we have to use the keyword *global*. However, this is not the case with mutable objects. Mutable objects can be modified without using the keyword *global*. We can change/modify the values of mutable objects such as that of dictionaries from inside the function without declaring them global.

```
thug = {'name': ' '}

def commision(thug_name):
    thug['name'] = thug_name

print(thug)
commision('showbaz')
print(thug)
```

```
{'name': ' '}
{'name': 'showbaz'}
```

```
thugs = ['musharaf', 'zardari']

def commision(accused):
    thugs.append(accused)

commision('nawaz')
thugs
```

```
['musharaf', 'zardari', 'nawaz']
```

```
thugs = set(thugs)

def commision(accused):
    thugs.add(accused)

commision('bajwa')
thugs
```

```
{'bajwa', 'zardari', 'nawaz', 'musharaf'}
```

Although tuples are immutable, thus they can not be modified but they can contain mutable objects such as lists, thus we can change/modify such contents of tuples from inside the function without using the keyword *global*.

```
thugs_tuple = (list(thugs),)

def commision(accused):
    thugs_tuple[0].append(accused)

print(thugs_tuple)
commision('pervailz elahi')
print(thugs_tuple)
```

```
(['bajwa', 'zardari', 'nawaz', 'musharaf'],)
(['bajwa', 'zardari', 'nawaz', 'musharaf', 'pervailz elahi'],)
```

thugs_tuple is a tuple but its first element is a list. We have modified the list inside the tuple without using the keyword `global`.

What will happen if we do assignment to a global variable inside function/local scope. The variable inside the local scope will be newly created while the immutable object outside the function will remain same.

```
thugs = ['musharaf', 'zardari']

def commision(accused):
    thugs = [accused]
    print(thugs, 'inside')
    return

print(thugs, 'outside')
commision('nawaz')
print(thugs, 'outside')
```

```
['musharaf', 'zardari'] outside
['nawaz'] inside
['musharaf', 'zardari'] outside
```

However, as said earlier, if we used the keyword `global`, this will affect the variable from global scope.

```
thugs = ['musharaf', 'zardari']

def commision(accused):
```

(continues on next page)

(continued from previous page)

```
    global thugs
    thugs = [accused]
    print(thugs, 'inside')
    return

print(thugs, 'outside')
commision('nawaz')
print(thugs, 'outside')
```

```
['musharaf', 'zardari'] outside
['nawaz'] inside
['nawaz'] outside
```

In general: variables in python are local unless declared otherwise.

Questions

Answer the following questions without running the code above them.

```
def foo():
    number = 2
    return number
```

Question: What value will be printed by the following code?

```
foo()
print(number)
```

Question: Did you get any error in the above code? If yes, then explain the error?

```
num = 1

def add_something(var):
    return var + num

def multipy_something(var):
    return var * num

a = add_something(5)
b = multipy_something(a)
```

Question: What will be the value of `b` in above code?

```
num = 1

def add_something(var):
    return var + num

def multipy_something(var):
    return var * num

```

(continues on next page)

(continued from previous page)

```
a = add_something(5)
num = 12
b = multipy_something(a)
```

Question: What will be the value of b in above code and why?

```
num = 1

def add_something(var):
    num = 14
    return var + num

def multipy_something(var):
    return var * num

a = add_something(5)
num = 12
b = multipy_something(a)
```

Question: What will be the value of b in above and why?

```
num = 1

def add_something(var):
    global num
    num = 14
    return var + num

def multipy_something(var):
    return var * num

a = add_something(5)
b = multipy_something(a)
```

Question: What will be the value of b in above code and why?

```
num = 1

def add_something(var):
    global num
    num = 14

def multipy_something(var):
    return var * num

add_something(5)
b = multipy_something(12)
```

Question: What will be the value of *b* in above code and why?

```
num = 1

def add_something(var):
```

(continues on next page)

(continued from previous page)

```
    global num
    num = 14

def multipy_something(var):
    num = 12
    return var * num

add_something(5)
num = num + 92
multipy_something(12)
```

```
144
```

Question: What will be the value of num in above code and why?

```
num = 1

def add_something(var):
    global num
    num = 14

def multipy_something(var):
    global num
    num = num * var

add_something(5)
num = num + 92
multipy_something(12)
```

Question: What will be the value of num in above code and why?

```
book = "black"

def change_book1():
    book = "white"
    return

def change_book2():
    global book
    book = book + " white"

change_book1()
book = book + " skin"
change_book2()
book = book + " masks"
```

Question: What will be the value of *book* in above code and why?

```
book = {'name': 'black skin white masks'}

def add_info():
    book['author'] = 'frantz fanon'
```

(continues on next page)

(continued from previous page)

```
    return

def change_info():
    book['name'] = 'white skin black masks'
    return

add_info()
change_info()
```

Question: What will be the value of `book['name']` in above code and why?

Total running time of the script: (0 minutes 0.014 seconds)

1.15 nested functions

A nested function is a function inside a function.

```
def percent(x, n):
    def number():
        factor = n / 100
        print(x * factor)

    number()


percent(120, 10)
```

```
12.0
```

It is pertinent to note that the function `number` was able to use the variable `x` and `n` which are from outer function `percent`.

In following example, the inner function is making use of variable `factor` which is defined inside the outer function `percent`.

```
def percent(x, n):
    factor = n / 100

    def number(xx):
        print(xx * factor)

    number(x)


percent(120, 10)
```

```
12.0
```

It is more clear now that inner function can make use of variables from outer functions.

In following example, the outer function returns the inner function which can be used later on.

```
def percent(n):
    factor = n / 100

    def number(xx):
        print(xx * factor)

    return number


ten_percent = percent(10)  # execution of percent is finished
ten_percent(150)  # variable `factor` is still remembered/used
```

```
15.0
```

When we called the `percent` function, it returned the inner function `number`. We assigned this inner function to a variable `ten_percent`. Now we can use this variable to call the inner function. The inner function still remembers the value of `factor` which was set when we called the outer function `percent`. Thus when we call `ten_percent(150)`, it is actually calling the inner function `number` with the value of `factor` set to 0.1.

```
two_percent = percent(2)
two_percent(150)
```

```
3.0
```

```
ten_percent(300)  # factor is 0.1
```

```
30.0
```

The variable `factor` is linked with the inner function and so with `ten_percent` and `two_percent`. Both `ten_percent` and `two_percent` have different values of `factor` associated with them. It is worth noting that, even when the execution of function `percent` is finished, its variable `factor` is still remembered by `ten_percent` and `two_percent`.We can find out which value of `factor` is associated with `ten_percent` and which value is associated with `two_percent`.

```
ten_percent.__closure__[0].cell_contents
```

```
0.1
```

```
two_percent.__closure__[0].cell_contents
```

```
0.02
```

Although we finished execution of function `percent` in previous cell, but upon executing `ten_percent(300)`, the function `number`, still remembers that what `factor` it has to multiply with. In this way we link a certain data with a function.

Question: Following the same method as above, define a function which returns 33% of a number. Then use this function to find 33% of 300.

```
def full_name(first_n):
    prefix = 'Mr. '
```

(continues on next page)

(continued from previous page)

```
    def family_name(fam_name):
        suffix = 'sahab'

        def kuniyat(z):
            return prefix + first_n + ' ' + fam_name + ' ' + z + ' ' + suffix

        return kuniyat

    return family_name
```

```
full_name('Abdus')('Sattar')('Edhi')
```

```
'Mr. Abdus Sattar Edhi sahab'
```

```
fam_name = full_name('Rashid')  # fam_name is intermediate function
```

```
Kuniyat = fam_name('Minhas')  # kuniyat is innermost function
```

```
Kuniyat('Shaheed')  # Kuniyat is innermost function
```

```
'Mr. Rashid Minhas Shaheed sahab'
```

Why use nested function

- They can be used to avoid global variables. In upper example, the variable `prefix`, can be set as global variable outside `full_name`, but in this case we have made data encapsulation and now the variable *prefix* is only available inside the function where it is actually needed. This is also called information hiding.
- To improve readability. If there is a tiny function that will only be invoked by the

outer function, then this will help us determine what the function is all about.

```
def print_name(name, gender):
    def add_prefix(s):
        if s == 'Male':
            return 'Mr.'
        else:
            return 'Mrs.'

    print(add_prefix(gender), name)


print_name('lalak jan', 'Male')

print_name('Fatima Jinnah', 'female')
```

```
Mr. lalak jan
Mrs. Fatima Jinnah
```

We could have easily put `add_prefix` outside the function in outer scope, but in current case when we will read the code, it becomes clear to us that the function `add_prefix` is used only inside `print_name`

```
def add_family_name(fam_name):
    def make_name(first):
        return first + fam_name

    return make_name


name_maker = add_family_name(' Minhas')
name_maker('Rashid')
```

```
'Rashid Minhas'
```

nonlocal statement

```
def square_after_adding_one(_in):
    x = _in

    def add_one():
        x += 1  # trying to re-assign value of variable `x` from outer scope
        return x

    val = add_one() ** 2
    print(val)
```

```
# uncomment following line
# square_after_adding_one(2)  # -> UnboundLocalError local variable 'x' referenced before␣
↪assignment
```

```
x = 0


def square_after_adding_one(_in):
    x = _in

    def add_one():
        nonlocal x
        x += 1
        return x

    val = add_one() ** 2
    print(val, 'squared after adding 1')


print(x, 'global ')
square_after_adding_one(2)
print(x, 'global ')
```

```
0 global
9 squared after adding 1
0 global
```

Above, the initial value of *x* is 0. Inside the function *square_after_adding_one*, we create a variable *x* with value 2. At this point this *x* is different from the global *x*. Then we call *add_one* which increments *x* by 1. Note that here we are using *nonlocal* keyword. This keyword is used to tell the function that the variable *x* is not local to the function *add_one* Therefore, we are using the variable *x* from the outer function *square_after_adding_one* whose value is 2. So, *x* is incremented by 1 and then squared. The output is 9. Once we are outside the function *square_after_adding_one*, the global *x* is still 0 because we have not touched the global *x* inside the function *square_after_adding_one*.

```
x = 0


def square_after_adding_one(_in):
    x = _in

    def add_one():
        global x
        x += 1
        return x

    val = add_one() ** 2
    print(val, 'squared after adding 1')
    print(x, 'in outer function')


print(x, 'global ')
square_after_adding_one(2)
print(x, 'global ')
```

```
0 global
1 squared after adding 1
2 in outer function
1 global
```

In above example, we are using *global* keyword instead of *nonlocal*. This means that we are using the global variable *x* inside the function *add_one*. Therefore, we are adding *1* to the global *x* whose value is *0*. So, *x* becomes *1* and then squared. Thus the value of `val` is *1*. There is another variable *x* which is inside the function *square_after_adding_one* whose value is *2*. This variable is not used inside the function *add_one*.

Total running time of the script: (0 minutes 0.006 seconds)

1.16 iterator vs iterable

Important: This lesson is still under development.

An `iterable` is any object which can be looped over. An `iterable` can be turned into `iterator` after applying `iter` function on it. An `iterator` is an object upon which `next` method can be applied. (Note: This is not a complete definition but without OOP, I suppose this will work)

```
provinces = ["Sindh", "Balochistan", "Janubi Pubjab", "Hazara", "Pakhtunkhwa"]
for prov in provinces:
    print(prov)
```

```
Sindh
Balochistan
Janubi Pubjab
Hazara
Pakhtunkhwa
```

```
# uncomment following line
# next(provinces)  # -> TypeError: 'list' object is not an iterator
```

As the error explicitly states, `provinces` is not `iterator`. so we can not apply `next` method on it.

However, we can convert it into `iterator` after applying `iter` method on it.

```
prov_iterator = iter(provinces)
next(prov_iterator)
```

```
'Sindh'
```

but what is `next` good for?

```
# uncomment following 2 lines

#for i in range(len(provinces)):
#    print(next(prov_iterator))
```

we get `StopIteration` exception after we have iterated through all the items of list

Let's create the iterator again and run this code.

```
prov_iterator = iter(provinces)
for i in range(len(provinces)):
    print(next(prov_iterator))
```

```
Sindh
Balochistan
Janubi Pubjab
Hazara
Pakhtunkhwa
```

Now try *next* on *prov_iterator* once again

```
# uncomment following line
# next(prov_iterator)
```

So we can get next item from an iterator until it has returned all items from the iterable.

```
print(type(prov_iterator))
```

```
<class 'list_iterator'>
```

Let's create an iterator from *tuple*

```
provinces = ("Sindh", "Balochistan", "Janubi Pubjab", "Hazara", "Pakhtunkhwa")
prov_iterator = iter(provinces)
print(type(prov_iterator))
```

```
<class 'tuple_iterator'>
```

Do we need a better way to check if an object is iterator or not?

isinstance

can also be used if some object is of a particular type or not.

```
isinstance(2.5, float)
```

```
True
```

```
isinstance(2.5, str)
```

```
False
```

```
from collections.abc import Iterator, Iterable
isinstance(prov_iterator, Iterable)
```

```
True
```

```
isinstance(prov_iterator, Iterator)
```

```
True
```

```
sequences = {
             list: ["Islamabad", "Tehran", "Istambul"],
             str: "Islamabad",
             tuple: ("Islamabad", "Tehran", "Istambul"),
             dict: {"Islamabad": '', "Tehran": '', "Istambul": ''},
             int: 2,
             float: 2.5
}

for _type, obj in sequences.items():
    print(isinstance(obj, _type))
```

```
True
True
True
True
True
True
```

```
for obj in sequences.values():
    print(isinstance(obj, Iterable), isinstance(obj, Iterator))
```

```
True False
True False
True False
True False
False False
False False
```

```
for obj in sequences.values():
    if isinstance(obj, Iterable):
        obj = iter(obj)
        print(isinstance(obj, Iterable), isinstance(obj, Iterator))
    else:
        print("{} can not be converted into iterator because it is not iterable".
↪format(obj))
```

```
True True
True True
True True
True True
2 can not be converted into iterator because it is not iterable
2.5 can not be converted into iterator because it is not iterable
```

The key takeaways from above two cells are following:

- So `list`, `str`, `tuple` and `dict` are iterables as such and not iterators.

However they can be converted into `iterators` after applying *next* method on it.

- Every `iterator` is also and `iterable` but all iterables are not iterators.
- In order to be able to create `iterator` from an object, it must be `iterable`

first. Floats and integers are not iterable so can not be converted into iterators as well.

As an `iterator` is also `iterable` so it can also be used on right side of `for` loop.

```
for prov in prov_iterator:
    print(prov)
```

```
Sindh
Balochistan
Janubi Pubjab
Hazara
Pakhtunkhwa
```

If you run the above cell again, it will not print anything because the iterator is already exhausted.

Not all methods that can be applied on an `iterable` can also be applied on `iterator`. For example the `len` method can not be applied on *iterator*.

```
# uncomment following line
# len(prov_iterator)
```

we can however find the number of items in an iterator after converting it into list

```
prov_iterator = iter(provinces)
len(list(prov_iterator))
```

```
5
```

The methods `next` and `iter` for iterator and iterable actually come from `__next__` and `__iter__` methods which reside inside corresponding class. Since OOP concepts are not discussed in this course, so these methods are also not discussed.

Total running time of the script: (0 minutes 0.006 seconds)

1.17 read/write operations

This lesson shows how to read/write files using python.

creating new file

If we want to create a new file we can make use of `open` function. The second argument here `w` defines that we would like to create a new file.

```
open("NewFile.txt", "w")
```

```
<_io.TextIOWrapper name='NewFile.txt' mode='w' encoding='UTF-8'>
```

The first argument to `open` must be complete path of the file. If only file name is given, it means the file will be created at current working directory.

The `open` function returns a file handler which can be useful.

```
new_file = open("NewFile.txt", "w")
print(type(new_file))
```

```
<class '_io.TextIOWrapper'>
```

Using the file handler, we can check whether a file is open or closed. if a file is open, we can write in it. If it is closed, we can not.

```
print(new_file.closed)
```

```
False
```

We must close the file once we are done working with a file.

```
new_file.close()

print(new_file.closed)
```

```
True
```

Instead of manually closing the file everytime, we can automatically close it using `with` keyword. `with` is called context manager which makes sure that whenever we are out of it, the file is closed.

```
with open("NewFile.txt", "w") as fp:
    pass

print(fp.closed)
```

```
True
```

Even if there is an error during writing/reading the file, the context manager makes sure that the file is closed despite the error.

```
# uncomment following lines
# with open("NewFile.txt", 'w') as fp:
#     fp.write("Bajwa chor hai")
#     raise NotImplementedError(f"You are not allowed to utter this.")
```

If we uncomment and run the above cell, it will result in error but we can confirm that the file is still closed. We can verify this using following statement

```
print(fp.closed)  # --> True
```

```
True
```

writing to a new file

If we want to write to a new file, we can make use of `open` function but with `w` as second argument.

Context manager also returns us a file handler which can be used to read/write the file

```
with open("NewFile.txt", "w") as fp:
    fp.write("My Name is Ali")
```

Here, `fp` is file handler, which can can use to modify file.

The `write` function is for writing a string. If we want to write a list of strings, we can make use of `writelines` function.

```
lines = ['His name was Ali', 'He was very brave']

with open("NewFile.txt", "w") as fp:
    fp.writelines(lines)
```

If we write something to an existing file and open the file with `w`, the previous file will be overwritten. This can also cause loss of data.

```
new_lines = ['His name was Hussain.\n', 'He was very brave\n']

with open("NewFile.txt", "w") as fp:
    fp.writelines(new_lines)
```

If we see the *NewFile.txt* we will see that it has *new_lines* and not `lines. This is because the previous file was deleted altogether.

writing to already existing file

If want to write to an already existing file, the second argument to `open` function must be `a`

```
lines = ['He was killed in 661']

with open("NewFile.txt", "a") as fp:
    fp.writelines(lines)
```

writing with specific separator

Consider that we want to write following data into a file.

```
lines = ["1 2 3", "1 2 3", "1 2 3"]
```

We can do this using following lines of code.

```
with open("NewFile", 'w') as fp:
    for line in lines:
        line = ','.join(line.split())
        fp.write(line + '\n')
```

Even though we have not defined the extension of file i.e., the file name is *NewFile*, but this file is still a text file which can be opened by any text editor. Above we used comma `,` to separate each value in a line. However, we can use any other separator that we wish e.g. a tab.

```
with open("NewFile", 'w') as fp:
    for line in lines:
        line = '\t'.join(line.split())
        fp.write(line + '\n')
```

reading a file

If we want to read a file, the second argument to `open` function must be `r`. However, the file must exist at the location which is specified.

```
text = """Nb  C  O Cr
 1.0
     9.461699    0.0    0.0
     0.590249    0.31933   29.99
 Nb  C   O   Cr
  18   9  18   1
Cartesian
  0.13  1.87 11.074
 -1.44  4.60 11.076
 -3.02  7.33 11.075
  3.28  1.85 11.040"""

with open('NewFile', 'w') as fp:
    fp.write(text)
```

(continues on next page)

(continued from previous page)

```
lines = []
with open('NewFile', 'r') as fp:
    for line_num, line in enumerate(fp.readlines()):
        if line_num > 6:
            line = line.split()  # split the line based upon spaces and put all members␣
↪into a list
            lines.append([float(num) for num in line])

print(lines)
```

```
[[0.13, 1.87, 11.074], [-1.44, 4.6, 11.076], [-3.02, 7.33, 11.075], [3.28, 1.85, 11.04]]
```

We are using `readlines` function for reading all the lines lin the file. The `readlines()` actually returns us a `list` on which we can iterate. Then we are iterating over these lines one by one. Next we are saving the lines in *lines* list after line number 7. Above the first argument is only file name. This means that the file must exists in current folder (working directory).

reading large files

The problem with above methodology is that it reads all the lines into memory. If however, the file is large (in Giga-Bytes), we may not wish to read whole file into memory. In that case we can read line by line.

```
with open('NewFile', 'r') as fp:
    for idx, line in enumerate(fp):
        pass
```

At every iteration, the previous `line` from memory is overwritten by the new `line`.

writing to a specific line

If we want to write to a specific line in a file or modify a specific line in a file, we can achieve this by reading the whole lines and then adding/changing those specific lines and then writing the modified lines back to the same file.

```
with open('NewFile', 'r') as fp:
    lines = fp.readlines()

lines.insert(9, '1.111 2.2222 3.33333\n')  # 9 means line number 10

with open('NewFile', 'w') as fp:
    fp.writelines(lines)
```

writing json format

json is a human readable file format. This file format is similar to python dictionary.

```
import json

data = {"name": "Ali", "age": 63, 'is_bold': True}

with open("NewFile.json", "w") as fp:
    json.dump(data, fp)
```

The data that we wrote above, was a dictionary but we can write any other data to json file as far as it is of native python type.

```
with open("NewFile.json", "w") as fp:
    json.dump([1, 2, 3], fp)
```

By setting the `indent` keyword argument, we can make sure that all the data is not saved on a single line. This makes the json file more readable.

```
with open("NewFile.json", "w") as fp:
    json.dump(data, fp, indent=True)
```

We can sort the keys of saved dictionary in json file by setting `sort_keys` to True.

```
with open("NewFile.json", "w") as fp:
    json.dump(data, fp, indent=True, sort_keys=True)
```

However, the json file can save only native python types. If the data is not native python type, we get the `TypeError`

```
import numpy as np

np_data = np.array([2])

# uncomment following two lines, they will return in TypeError
# with open("jsonfile.json", "w") as fp:
#     json.dump(np_data, fp)  # -> TypeError: Object of type ndarray is not JSON␣
↪serializable
```

The error message very explicit says that the data we are trying to save is ndarray and it can not be serialized .

We can verify this by checking the type of *data*.

```
print(type(np_data))
```

```
<class 'numpy.ndarray'>
```

data is an array, which means it consists of multiple values. We can get the first value of *data* by slicing it using slice operator `[]`.

```
np_data_0 = np_data[0]

print(np_data_0)
```

```
2
```

Now If we try to save it in json file,

```
# Uncomment following two lines, they will result in TypeError
# with open("jsonfile.json", "w") as fp:
#     json.dump(np_data_0, fp)  # -> TypeError: Object of type int32 is not JSON␣
↪serializable
```

The above error message says that `int32` is also not serializable. This is because the first member of `data` is `int32` type which is from numpy library.

```
print(type(np_data_0))
```

```
<class 'numpy.int64'>
```

This is because `int32` is also not python's native type but is from numpy library.

We can convert `int32` into python's `int` type and then we can save it into json file format.

```
np_data_0_int = int(np_data_0)

print(type(np_data_0_int))
```

```
<class 'int'>
```

```
with open("NewFile.json", "w") as fp:
    json.dump(np_data_0_int, fp)
```

```
np_data = np.array([2, 3, 4])

# Uncomment following two lines, they will result in TypeError
# with open("jsonfile.json", "w") as fp:
#     json.dump(np_data, fp)
```

The `tolist` method of numpy array converts the numpy array into list which is python native type and can be saved as json.

```
data_list = np_data.tolist()

with open("NewFile.json", "w") as fp:
    json.dump(data_list, fp)
```

reading json format

In order to read the json file, we can make use of `json.load()` function. The first argument must be file path.

```
with open("NewFile.json", "r") as fp:
    data = json.load(fp)

print(data)
```

```
[2, 3, 4]
```

The type of the data is preserved when we load the json file.

```
print(type(data))
```

```
<class 'list'>
```

writing in binary format

Above when we wrote the data, the saved file was human readable. This means that you can open the file and see/read the data. However, there is a cost of doing this. If the data is large, the file size gets extremely large and the process of reading and writing becomes slow. This can be avoided by writing the data into binary format. The downside here is that the written file is not human readable unless you have specific software e.g HDFView . These software actually convert the binary data into human readable and show it. Saving the data into binary format is a very large topic and there are many built-in and third-party libraries in python for it. However, here we will only cover basics of `pickle` module of python.

```
import pickle
```

When we want to save the data into binary format/file, the second argument to `open` function must be `wb`. Here, `w` means that we are creating a new file and `b` represents that the data will be written in binary format.

```
my_bytes = [120, 3, 255, 0, 1000]
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)
```

Above all the elements in list were integer, however they can also be float

```
my_bytes = [120, 3, 255, 0, 1000.0]
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)
```

also string data can be saved as binary using pickle module.

```
my_bytes = [120, 3, 255, 0, 1000.0, 'a']
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)
```

Similarly we can write `tuple` or `dictionary` data into binary format.

```
my_bytes = [120, 3, 255, 0, 1000.0, 'a', (1, 2), None]
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)
```

```
my_bytes = [-1200, 3, 255, 0, 1000.0, 'a', {'a': 1}, True]
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)
```

reading binary format

If we want to read binary file, we can use `rb` keyword in `open` function as second argument.

```
with open("NewFile", "rb") as my_pickle_file:
    my_bytes = pickle.load(my_pickle_file)

print(my_bytes)
```

```
[-1200, 3, 255, 0, 1000.0, 'a', {'a': 1}, True]
```

The pickle module can read/write a wide range of data types.

```
my_bytes = np.array([120, 3, 255, 0, 1000.0])
with open("NewFile", "wb") as my_pickle_file:
    pickle.dump(my_bytes, my_pickle_file)

with open("NewFile", "rb") as my_pickle_file:
    my_bytes = pickle.load(my_pickle_file)

print(type(my_bytes))
```

```
<class 'numpy.ndarray'>
```

Above we wrote numpy data type which is not python's native data type. Moreover, when we read binary file, we still got numpy data type.

Total running time of the script: (0 minutes 0.018 seconds)

1.18 Exceptions

This lesson shows the usage of exceptions in python

An exception is basically an error which is raised when a statement or command fails in python.

```
a = 2

assert isinstance(a, int)
```

The function `isinstance` is being used here to check the `type` of a python object. The above statement did not return anything because the assertion was successful. However if the assertion fails, we get a special kind of error called `AssertionError`.

AssertionError

If we run the code given below, it will result in `AssertionError` because the variable *a* is `int` type and not `float` type.

```
# uncomment following 1 line
# assert isinstance(a, float)
```

We can write more useful error messages instead of throwing just `AssertionError` as follows,

```
# uncomment following 1 line
# assert isinstance(a, float), f"a is not float but is of {type(float)} type"
```

Similar to `AssertionError`, there are many other types of errors/exceptions in python

TypeError

This error is raised when an operation is failed due to wrong `type` of a variable.

```
# uncomment following line
# 4 + 'a'  # -> TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Since we can not add a string into an integer, we got `TypeError` above.

ModuleNotFoundError

This error is raised when a specific module that we are importing is not **available/installed**.

```
# uncomment following 1 line
# import ai4water  # -> ModuleNotFoundError: No module named 'ai4water'
```

Since `ai4water` package is not installed, we got `ModuleNotFoundError` above.

Whenever we import a variable, function or class or any python object, python searches in all the directories (folders) which are in its `path`. So when we say a package is not installed or available, it means, this package or the object that we are importing is not available in any of the directories in the `path`. In other words we also say that, this package/object is not available in python's path. If we want to check what are the lists of directories/folders in python's path, we can do this by printing the value of `sys.path`.

sys.path returns a `list` so we can iterate over it.

```
import sys

for p in sys.path:
    print(p)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/docs/
↪source
/home/docs/.asdf/installs/python/3.9.19/lib/python39.zip
/home/docs/.asdf/installs/python/3.9.19/lib/python3.9
/home/docs/.asdf/installs/python/3.9.19/lib/python3.9/lib-dynload
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/envs/latest/lib/python3.9/
↪site-packages
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪basics
```

KeyError

This error is raised when a specific key is not available in a dictionary.

```
human = {"name": "Ali"}

# uncomment following 1 line
# print(human['age'])  # KeyError: 'age'
```

Since the dictionary *human* does not have *age* key, we will get `KeyError`, if we will run the above cell.

PermissionError

This error is raised when the user does not have permission to access the file or to do an operation on such a file for which he/she does not have corresponding rights.

```
f = open('NewFile.txt', 'w')

import os

# uncomment the following line
# os.remove('NewFile.txt')  # -> PermissionError
```

since the file *NewFile.txt* was already opened, we can not delete it and got `PermissionError`.

```
f.close()
os.remove('NewFile.txt')
```

Once we close the file using `f.close()`, then we can delete the file.

`PermissionError` is also raised when you are trying to open/create a file but the first argument to `open` function is folder instead of file.

FileNotFoundError

This error is raised when the the file that we are trying to open does not exist in the directory/path.

```
# uncomment following 1 line
# open("NonExistingFile", "r")  # -> FileNotFoundError: [Errno 2] No such file or␣
↪directory: 'NonExistingFile'
```

IndexError

This error is raised when try to access a value from a sequence based upon the index which is not available for that sequence. For example, for a list with three members, if we try to access its fourth member, we will get `IndexError`.

```
my_list = [1, 2, 3]
# uncomment following line
# my_list[3]  # -> IndexError: list index out of range
```

Same is true for tuples.

```
my_tuple = (1, 2, 3)

# uncomment following line
# my_tuple[3]  # -> IndexError: tuple index out of range
```

There is another commonly encountered error i.e., `AttributeError`. We are not introducing it here. It will discussed later in *3.15 __getattr__*.

Error handling

So what is the purpose of knowing all of these errors?

Sometimes we would like to forecast/predict an error and bypass the error by applying the solution. In the function below, we would like to add a value in 2. However, if due to `TypeError` we can not do so, then we would first convert it to integer before adding it into 2.

```
def add_number(val):
    try:
        val = val + 2
    except TypeError:
        print('TypeError was encountered but bypassed')
        val = int(val) + 2
    return val


add_number(2)
```

```
4
```

```
add_number('2')
```

```
TypeError was encountered but bypassed

4
```

Sometimes we want to handle/catch multiple exceptions. We can do this in one line by writing all the exceptions inside a tuple after `except` keyword.

```
def multiple_exceptions(wname):
    try:
        out = float(wname.split('_')[2])
    except (ValueError, IndexError) as e:
        raise Exception(f"{wname} raised ", e)
    return out
```

Above, we want to catch ValueError and IndexError at the same time.

```
# uncomment following line
# multiple_exceptions("11_2.5.h5")  # -> IndexError
```

```
# uncomment following line
# multiple_exceptions("11_2_5.h5")  # -> ValueError
```

```
multiple_exceptions("11_2_5")
```

```
5.0
```

Error handling is also used by printing out more useful error messages to the user. Suppose the package `ai4water` is not installed which may be required. Then instead of just throwing `ModuleNotFoundError`, we can help the user by giving more helpful error message

```
def better_error_message():
    try:
        import ai4water
    except ModuleNotFoundError as e:
        raise ModuleNotFoundError(f"You must install ai4water using pip install ai4water␣
↪\n{e}")
    return

# uncomment following line
# better_error_message()
```

Custom Errors

We can create our own errors based upon our needs. Any new error must inherit from `Exception` class or its base classes. If at this stage you are uncomfortable with the concept of `class` or *base class*, you can jump to lessons of *3.1 introduction* and *3.13 inheritance* in *3. oop* chapter to know about them.

Below we create an error which will be raised if a value is above 256.

```
class IllegalValue(Exception):
    def __init__(self, val):
        self.val = val

    def __str__(self):
        return f"val must be below 256 but it is {self.val}"


value = 1000
```

We can invoke this as given below.

```
# Uncomment following two lines
# if value>256:
#     raise IllegalValue(value)
```

Note

Error messages are blessings in disguise. Whenever you get an error, you should consider yourself very lucky. Most of the learning happens in the process of understanding and resolving the errors. Without errors (and resolving them), the process of coding will look like copy paste and anything you learn will be forgotten very soon. So don't be afraid of errors. Embrace them with open heart and learn from them.

Total running time of the script: (0 minutes 0.004 seconds)

1.19 generators

This lesson explains the concept of generators in python and the use of keyword `yield`.

We know that we can do list comprehension as below,

```
print([i for i in range(10)])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Similarly we can also do tuple comprehension

```
print((i for i in range(10)))
```

```
<generator object <genexpr> at 0x7f13697a8e40>
```

However, we see that a tuple comprehension returns a generator.

```
a = (i**i for i in range(10))
```

We can unpack generators using * as following

```
print(*a)
```

```
1 1 4 27 256 3125 46656 823543 16777216 387420489
```

iterating over generator

We can use generators in a for loop because we can iterate over them.

```
a = (i for i in range(10))

for i in a:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

The `next` function on a generator is akin to running the for loop for one iteration. It means we want to get the next value from generator.

```
a = (i**i for i in range(10))
next(a)
```

```
1
```

```
next(a)
```

```
1
```

```
next(a)
```

```
4
```

We can call the `next` function as long as the function is not exhausted.

Running a for loop on a generator actually calls `next` function on the generator until it is exhausted.

```
for i in a:
    print(i)
```

```
27
256
3125
46656
823543
16777216
387420489
```

Since all the iterations are complete, our generator is now exhausted. If we try to get next value from generator, it will throw an error.

```
# uncomment the following line
# next(a)
```

```
for i in a:
    print(i)
```

The above for loop was run on exhausted generated, therefore no iteration was run. However, it should be noted that StopIteration exception was not raised by for loop. This is because, for loop automatically detects the StopIteration and stop its iteration instead of raising the error/exception.

Above, when we created generator using tuple comprehension, we were just returning `i` but we can do a complicated or computationally heavy stuff e.g. reading a file at each iteration.

```
def read_file(idx):
    print(f"reading file {idx}")
    return idx


reader = (read_file(i) for i in range(10))

for _ in reader:
    pass
```

```
reading file 0
reading file 1
reading file 2
reading file 3
reading file 4
reading file 5
reading file 6
reading file 7
reading file 8
reading file 9
```

or a memory intensive computation at each step.

```
a = (i**i for i in range(10))

for i in a:
    print(i)
```

```
1
1
4
27
256
3125
46656
823543
16777216
387420489
```

Working of generator

A generator actually breaks the computation flow in a for loop and executes the commands inside the for loop one by one. This helps in executing the memory intensive work one by one instead of executing all the code at once. For example, instead of reading all the 10 files at once and then processing them, we read one file at one time, process it and then read the next file.

yield

How to generate a generator using `yield` keyword?

```
def read_files(num_files):
    file_content = []
    for f in range(num_files):
        _file_content = read_file(f)
        file_content.append(_file_content)
    return


read_files(10)
```

```
reading file 0
reading file 1
reading file 2
reading file 3
reading file 4
reading file 5
reading file 6
reading file 7
reading file 8
reading file 9
```

Above we are reading all the files and saving their content in a list at once. If the files are large and we don't need all the files at once, then we would like to read one file, use/process its contents and then read the next file. In such a case, we would like to write a function, which reads one file at a time. This can be accomplished by using `yield` keyword.

```
def read_files(num_files):
    for f in range(num_files):
        yield read_file(f)


reader = read_files(10)

print(type(reader))
```

```
<class 'generator'>
```

Let's iterate through the generator

```
for _ in reader:
    pass
```

```
reading file 0
reading file 1
reading file 2
reading file 3
reading file 4
reading file 5
reading file 6
reading file 7
reading file 8
reading file 9
```

We can not have any statement in a function after `return` keyword. However, we can have statements after `yield`. These statements are executed at the next iteration.

```
def read_files(num_files):
    print("entering read_files function")
    for f in range(num_files):
        print(f"at iteration {f}")
        yield read_file(f)
        print(f"at iteration {f} after yield")
```

(continues on next page)

(continued from previous page)

```
reader = read_files(10)

next(reader)
```

```
entering read_files function
at iteration 0
reading file 0

0
```

We see that the string **at iteration {f} after yield** is not printed yet. This is because only first iteration of for loop is complete. However, this string will be printed at the start of next iteration.

```
next(reader)
```

```
at iteration 0 after yield
at iteration 1
reading file 1

1
```

When a function has `yield` keyword, we can have statements even outside the for loop. All the statements outside the for loop will be executed after the last iteration of the generator.

```
def read_files(num_files):
    print("entering read_files function")
    for f in range(num_files):
        print(f"at iteration {f}")
        yield read_file(f)
        print(f"at iteration {f} after yield")
    print('end of for loop')
    return


reader = read_files(10)

print(type(reader))
```

```
<class 'generator'>
```

```
for _ in reader:
    pass
```

```
entering read_files function
at iteration 0
reading file 0
at iteration 0 after yield
at iteration 1
reading file 1
at iteration 1 after yield
at iteration 2
```

(continues on next page)

(continued from previous page)

```
reading file 2
at iteration 2 after yield
at iteration 3
reading file 3
at iteration 3 after yield
at iteration 4
reading file 4
at iteration 4 after yield
at iteration 5
reading file 5
at iteration 5 after yield
at iteration 6
reading file 6
at iteration 6 after yield
at iteration 7
reading file 7
at iteration 7 after yield
at iteration 8
reading file 8
at iteration 8 after yield
at iteration 9
reading file 9
at iteration 9 after yield
end of for loop
```

We wee that the string *end of for loo* is printed only once at the end.

yield from

Consider the case where we have a list which further consists of one or more lists

```
my_list = [1, 2, [3, 4], 5]
```

If we want to yield one element from this list by flattening it, the naive approach would be following

```
def flatten(elements):
    for elem in elements:
        if isinstance(elem, list):
            for _elem in elem:
                yield _elem
        else:
            yield elem


flattener = flatten(my_list)

for i in flattener:
    print(i)
```

```
1
2
```

(continues on next page)

(continued from previous page)

```
3
4
5
```

Above we iterate over each value (elem) of my_list. Whenever, `elem` is itself a list, we make another `for` loop.

However, we can achieve the same thing using `yield from` keyword.

```
def flatten(elements):
    for elem in elements:
        if isinstance(elem, list):
            yield from elem
        else:
            yield elem
```

Above, instead of writing another `for` loop for `elem`, we are using the keyword `yield from`.

```
flattener = flatten([1, 2, [3, 4], 5])

for i in flattener:
    print(i)
```

```
1
2
3
4
5
```

What if we have list inside list of another list or a variation of such nested lists?

```
flattener = flatten([1, 2, [3, 4], 5, [6, [7, 8]]])

for i in flattener:
    print(i)
```

```
1
2
3
4
5
6
[7, 8]
```

`[7,8]` is printed in same line, which means this inner list is not flattened by our function.

We may be tempted to add another `if` statement for checking whether any member in `elem` is a list or not. However, we can call the `flatten` function from inside so that the recursion continues until we flatten the list to its last/innermost member.

```
def flatten(elements):
    for elem in elements:
        if isinstance(elem, list):
            yield from flatten(elem)  # we call the function again
```

(continues on next page)

(continued from previous page)

```
        else:
            yield elem


flattener = flatten([1, 2, [3, 4], 5, [6, [7, 8]]])

for i in flattener:
    print(i)
```

```
1
2
3
4
5
6
7
8
```

Now the list is flattened to its innermost member.

```
flattener = flatten([1, 2, [3, 4], 5, [6, [7, [8, 9, 10]]]])

for i in flattener:
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Total running time of the script: (0 minutes 0.011 seconds)

1.20 installing packages

What is meaning of installing a python package?

```
def foo():
    pass

foo()
```

Above we used the function *foo* which was defined above in the same file where it was called. However, if the *foo* was defined not in the same file, but in some other file, we would have to `import` the *foo* function from that file. That file containing *foo* function can be located anywhere in the hard drive/storage of our machine/computer. If python is able to 'see' that file and consequently the foo function in that file, you can import foo from that file. But if python is not able to see that file, even if it is located somewhere in your computer, you can not import that function. This applies not only

to functions by to every python object like variables, classes etc. The process of putting the required files in required format at the required location can be termed as installing a library/module/package in python. When we say that we want to install a library say `numpy` in python, it means we want all the files of this library at a place in our computer so that python can see it. Moreover, the files should be in such a form that we can import our desired functions from them.

putting module in cwd

A simpler solution of installing a package/library is to simply put in our current working directory. This is because if we print `sys.path`, the first directory is the current working directory. This means, whenever we try to import something, python will look into current working directory. Therefore, if we put some file/package/library in our current working directory, we can simply import that library without any problem.

using site

Putting a lot of libraries/packages in our current working library can clutter current working direcotry. This is because we may need hundred of libraries to work with. One solution is to place those packages at any place in our desired location in our machine, and then import that package from that location. We can add the location of that package into python's `path` during runtime using `site.addsitedir` function.

```
import site
site.addsitedir('path/to/my/package')
```

pip

An easy and one of the most commonly used tool to install a package is using `pip` command. We can do this in the terminal and not in python console. However, since ipython console allows us to run terminal commands by prefixing them with `!`, we can also use `pip` in ipython console. Remember that `pip` installs the package in `site_packages` directory of python. Also note that the built-in python modules can not be installed using `pip`. Following examples show how to use `pip` to install a package called `easy_mpl`.

```
pip install easy_mpl
```

```
pip install easy_mpl==0.21.3
```

```
pip install -update easy_mpl
```

```
pip install easy_mpl[all]
```

conda

```
# -e .
```

Total running time of the script: (0 minutes 0.001 seconds)

1.21. builtin functions

This lesson shows the usage of builtin functions available in python. These functions are available in python without importing any module. These are not all the builtin functions but only those which are used frequently.

Important: This lesson is still under development.

slice

```
a = [1,2,3,4,5,6,7,8,9,10]
print(a[slice(2)])
```

```
[1, 2]
```

```
print(a[slice(2, 8)])
```

```
[3, 4, 5, 6, 7, 8]
```

```
print(a[slice(2, 8, 2)])
```

```
[3, 5, 7]
```

```
a = "This is a string"
print(a[slice(2)])
print(a[slice(2, 8)])
print(a[slice(2, 8, 2)])
```

```
Th
is is
i s
```

```
a = (1,2,3,4,5,6,7,8,9,10)
print(a[slice(2)])
print(a[slice(2, 8)])
print(a[slice(2, 8, 2)])
```

```
(1, 2)
(3, 4, 5, 6, 7, 8)
(3, 5, 7)
```

```
a = {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
# print(a[slice(2)])
```

Indeed, any python object which can be sliced using `[]` operator, can also be sliced using `slice` function. Although, `[]` operator and `slice` function look similar however they differ in their behavior. `[]` operator returns the values whereas `slice` function returns a `slice` object. Moreover, `slice` function is more readable and flexible.

zip

zip is used to iterate over two or more than two sequences.

```
for i,j in zip([1,2,3], [11, 12, 13, 14]):
    print(i, j)
```

```
1 11
2 12
3 13
```

any

```
vals = [1,2,3]
print(any([val>3 for val in vals]))
```

```
False
```

```
vals = [1,2,3, 4]
print(any([val>3 for val in vals]))
```

```
True
```

all

```
vals = [1,2,3,4,5]
print(all([isinstance(val, int) for val in vals]))
```

```
True
```

```
vals = [1,2,3,4,5.0]
print(all([isinstance(val, int) for val in vals]))
```

```
False
```

```
vals = [1,2,3,4,5.0]
print(all([isinstance(val, (int, float)) for val in vals]))
```

```
True
```

```
vals = [1,2,3,4,'5.0']
print(all([isinstance(val, (int, float)) for val in vals]))
```

```
False
```

Question:

What is the `type` of the output returned by `any` and `all` functions?

sorted

```
vals = ['a', 'b', 'c']
for val in sorted(vals):
    print(val)
```

```
a
b
c
```

```
vals = ['a', 'c', 'b']
for val in sorted(vals):
    print(val)
```

```
a
b
c
```

```
vals = [1,5,3,10.0]
for val in sorted(vals):
    print(val)
```

```
1
3
5
10.0
```

reversed

```
vals = ['a', 'b', 'c']
for val in reversed(vals):
    print(val)
```

```
c
b
a
```

enumerate

```
vals = ['a', 'b', 'c']
for idx, val in enumerate(vals):
    print(idx, val)
```

```
0 a
1 b
2 c
```

Question:

What will be the output of the following code?

```
names = ['jamaludin', 'zaynaldin', 'nurullah shustari', 'kamil dehlavi', 'baqir sadr]
years = ['1385', '1558', '1610', '1809', '1980']
for (idx,name), year in zip(enumerate(names), years):
    print(idx, name, year)
```

Question:

Consider the following list

```
dob_years = ['1334', '1506', '1542', '1700s', '1935']
```

Now modify the `for` loop in the previous example to also iterate over `dob_years` list along with `names` and `years`.

map

Suppose we have function which takes an input and returns square of it

```
def square(x):
    return x**2
```

Now if we want to call this function on several values, we can make a list of all the values on which we want to call it

```
vals = [1,2,3,4,5]
```

and then we can call this function in a loss

```
for val in vals:
    print(square(val))
```

```
1
4
9
16
25
```

An alternative to calling the function in an explicit for loop is to make use of `map` function.

```
mapper = map(square, vals)
```

The `map` function returns an iterator which we can be converted into a `list`

```
print(type(mapper))
```

```
<class 'map'>
```

```
from collections.abc import Iterator
print(isinstance(mapper, Iterator))
```

```
True
```

We can extract all values from iterator by calling `list` method on it.

```
list(mapper)
```

```
[1, 4, 9, 16, 25]
```

we can also provide any builtin function as first argument to `map`

```
list(map(float, vals))
```

```
[1.0, 2.0, 3.0, 4.0, 5.0]
```

we can also use map with functions which take multiple input input arguments.

```
def power(x, n):
    return x**n

vals1 = [1,2,3,4]
vals2 = [1,2,3,4]
list(map(power, vals1, vals2))
```

```
[1, 4, 27, 256]
```

Map has several advantages over an explicit for loop e.g

- It is fast since it is written in C
- It is memory efficient as it returns an iterator

Question:

Convert the years in following list from Hijri to Gregorian[1] calenden using `map` function

```
hijri_years = [40, 50, 61, 95, 114, 148, 183, 203, 220, 254, 260]
```

Total running time of the script: (0 minutes 0.010 seconds)

2.7.2 2. builtin modules

This chapter describes some useful built-in modules/packages in python.

2.1 os

Important: This lesson is still under development.

When python is installed, several modules/packages are pre-installed in it. These modules provide some very basic functionality to the user. The `os` is such a built-in module which provides the functionalities about paths or operating system.

```
import os
```

[1] https://github.com/dralshehri/hijridate

system()

It is used to execute commands that we would otherwise execute using command prompt on windows or on terminal. For example on Windows we can do `os.system(dir)` where `dir` is a command for Windows command prompt

```
os.system(dir)
```

getcwd()

It returns the current working directory.

```
print(os.getcwd())
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules
```

```
print(type(os.getcwd()))
```

```
<class 'str'>
```

path

The `path` submodule of `os` module contains several helpful functions to manipulate path.

Find out whether the path *p* exists or not

```
p = os.getcwd()
os.path.isdir(p)
```

```
True
```

Find out whther *p* is a file or not

```
os.path.isfile(p)
```

```
False
```

—

```
# Find the parent directory of `p`.
os.path.dirname(p)
```

```
'/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts'
```

—

```
os.path.abspath(p)
```

```
'/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules'
```

```
os.path.split(p)
```

```
('/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts
↪', 'builtin_modules')
```

—

```
os.path.basename(p)
```

```
'builtin_modules'
```

—

```
os.path.exists(p)
```

```
True
```

remove()

delete a file from disk

replace()

listdir

Returns a list of folders/directories in a given path.

```
os.listdir(p)
```

```
['readme.txt', '_os.py', '_types_typing.py', '_csv.py', '_warnings.py', '_random_module.
↪py', '_time.py', '_pathlib.py', '_sys.py', '_copy.py', '_collections.py', '_itertools.
↪py', '_math.py']
```

environ

```
environ = os.environ
print(environ)
print(type(environ))
print(len(environ))
```

```
environ({'READTHEDOCS_VIRTUALENV_PATH': '/home/docs/checkouts/readthedocs.org/user_
↪builds/python-seekho/envs/latest', 'READTHEDOCS_CANONICAL_URL': 'https://python-seekho.
↪readthedocs.io/en/latest/', 'HOSTNAME': 'build-26244417-project-717928-python-seekho',
↪'READTHEDOCS_GIT_CLONE_URL': 'https://github.com/AtrCheema/python-seekho.git', 'HOME':
↪'/home/docs', 'NO_COLOR': '1', 'READTHEDOCS': 'True', 'READTHEDOCS_PRODUCTION_DOMAIN':
↪'readthedocs.org', 'READTHEDOCS_REPOSITORY_PATH': '/home/docs/checkouts/readthedocs.
↪org/user_builds/python-seekho/checkouts/latest', 'READTHEDOCS_PROJECT': 'python-seekho
↪', 'READTHEDOCS_OUTPUT': '/home/docs/checkouts/readthedocs.org/user_builds/python-
↪seekho/checkouts/latest/_readthedocs/', 'PATH': '/home/docs/checkouts/readthedocs.org/
↪user_builds/python-seekho/envs/latest/bin:/home/docs/.asdf/shims:/home/docs/.asdf/bin:/
↪usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'READTHEDOCS_VERSION_TYPE
↪': 'branch', 'LANG': 'C.UTF-8', 'READTHEDOCS_LANGUAGE': 'en', 'DEBIAN_FRONTEND':
↪'noninteractive', 'READTHEDOCS_GIT_COMMIT_HASH':
↪'a06cdc71c8d1c55a8fb4a4c99a19ace59c70e4bf', 'READTHEDOCS_VERSION_NAME': 'latest',
↪'READTHEDOCS_VERSION': 'latest', 'PWD': '/home/docs/checkouts/readthedocs.org/user_
↪builds/python-seekho/checkouts/latest/docs/source', 'READTHEDOCS_GIT_IDENTIFIER':
↪'master', 'DOCUTILSCONFIG': '/home/docs/checkouts/readthedocs.org/user_builds/python-
↪seekho/checkouts/latest/docs/source/docutils.conf'})
<class 'os._Environ'>
22
```

wait()

rename()

renames()

mkdir()

creates a directory if all its upper/parent directories are present.

```
new_folder = os.path.join(os.getcwd(), "NewFolder")

print(os.path.exists(new_folder))
os.mkdir(new_folder)

print(os.path.exists(new_folder))
```

```
False
True
```

```
p = os.path.join(os.getcwd(), "NonExistentFolder", "InsideNonExistentFolder")

print(os.path.exists(p))
```

```
False
```

uncomment following line

```
# os.mkdir(p)  # FileNotFoundError
```

os.makedirs

```
os.makedirs(p)
print(os.path.exists(p))
```

```
True
```

rmdir()

```
print(os.path.exists(new_folder))
os.rmdir(new_folder)
print(os.path.exists(new_folder))
```

```
True
False
```

cpu_count()

Cound the number of cpus/cores/processes available.

```
print(os.cpu_count())
```

```
2
```

chdir()

```
original_wd = os.getcwd()
print("Current working directory is \n", original_wd)
print(f"It has {len(os.listdir(os.getcwd()))} sub-directories/folders in it.")
new_wd = os.path.join(os.path.dirname(original_wd))
os.chdir(new_wd)
print("Now we have changed working directory to \n", new_wd)
print(f"This new directory has {len(os.listdir(os.getcwd()))} sub-directories/folders in␣
↪it")
# now changing back to original
os.chdir(original_wd)
```

```
Current working directory is
 /home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules
It has 14 sub-directories/folders in it.
Now we have changed working directory to
 /home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts
This new directory has 8 sub-directories/folders in it
```

walk

```
for dirpath, dirnames, filenames in os.walk(os.getcwd()):
    print(dirpath, dirnames, filenames)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules ['NonExistentFolder'] ['readme.txt', '_os.py', '_types_typing.py', '_
↪csv.py', '_warnings.py', '_random_module.py', '_time.py', '_pathlib.py', '_sys.py', '_
↪copy.py', '_collections.py', '_itertools.py', '_math.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/NonExistentFolder ['InsideNonExistentFolder'] []
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/NonExistentFolder/InsideNonExistentFolder [] []
```

```
python_seekho_scripts = os.path.join(os.path.dirname((os.path.abspath(''))))
for dirpath, dirnames, filenames in os.walk(python_seekho_scripts):
    print(dirpath, dirnames, filenames)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts [
↪'numpy', 'basics', 'advanced', 'oop', 'pandas', 'builtin_modules', 'plotting'] [
↪'readme.txt']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪numpy [] ['readme.txt', 'stack_vs_concatenate.py', 'dimensions_axis.py', 'digitize.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪basics [] ['nested_functions.py', 'for_loops.py', 'installing_packages.py',
↪'conditional_statements.py', 'copying_lists.py', 'args_and_kwargs.py', 'readme.txt',
↪'exceptions.py', 'lists.py', 'while_loops.py', 'dictionaries.py', 'generators.py',
↪'NewFile.json', 'functions.py', 'iterator_vs_iterable.py', 'variables.py', 'operators.
↪py', 'sets.py', 'io_operations.py', 'global_vs_local.py', 'sequential_data.py',
↪'NewFile', 'keywords.py', 'print_function.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪advanced [] ['reg_expressions.py', 'daily_q.csv', 'readme.txt', 'interfacing_with_
↪fortran.py', 'parallel_processing.py', 'interfacing_with_c_cpp.py', 'unit_testing.py',
↪'read_csv_gpt.cpp', 'interfacing_with_matlab.py', 'interfacing_with_cpp.py', 'speeding_
↪up_with_numba.py', 'scripts_to_executables.py', '_cython.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪oop [] ['class.py', '__getattr__.py', 'readme.txt', 'init.py', 'inheritance.py', 'call.
↪py', 'class_vs_instance_attributes.py', 'attributes.py', 'str_repr.py', 'class_methods.
↪py', 'introduction.py', 'public_vs_private_attributes.py', 'magical_methods.py',
↪'methods.py', 'static_methods.py', 'getattr_setattr.py', 'descriptors.py', 'property.py
↪']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪pandas [] ['speeding_up.py', 'readme.txt', 'apply.py', 'pivot_vs_melt.py', 'multi-
↪indexing.py', 'groupby.py', 'working_with_timeseries.py', 'dataframe_vs_series.py',
↪'io.py', 'indexing_vs_slicing.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules ['NonExistentFolder'] ['readme.txt', '_os.py', '_types_typing.py', '_
↪csv.py', '_warnings.py', '_random_module.py', '_time.py', '_pathlib.py', '_sys.py', '_
↪copy.py', '_collections.py', '_itertools.py', '_math.py']
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/NonExistentFolder ['InsideNonExistentFolder'] []
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/NonExistentFolder/InsideNonExistentFolder [] []
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting [] ['_bar.py', 'readme.txt', '_plot.py', '_contour.py', 'marginal_plots.py',
↪'_scatter.py', '_circular_bar.py', '_parallel_plot.py', 'intro.py', '_hist.py', '_
↪dumbell.py', '_regplot.py', '_lollipop.py', '_ridge.py', '_boxplot.py', 'pie.py', '_
↪taylor.py', '_violin.py', '_imshow.py', '_spider.py']
```

finding files

Let's create few files first.

```
for i in range(5):
    with open(f'TextFile_{i}.txt', 'w'):
        pass
for i in range(5):
    with open(f'CsvFile_{i}.csv', 'w'):
        pass
```

If we want to find all files and folders within a specific path

```
path_to_look = os.getcwd()

print(os.listdir(path_to_look))
```

```
['NonExistentFolder', 'TextFile_1.txt', 'CsvFile_0.csv', 'readme.txt', 'TextFile_4.txt',
↪'_os.py', 'CsvFile_4.csv', '_types_typing.py', 'CsvFile_1.csv', '_csv.py', 'CsvFile_2.
↪csv', '_warnings.py', 'TextFile_2.txt', '_random_module.py', '_time.py', '_pathlib.py',
↪ '_sys.py', 'CsvFile_3.csv', 'TextFile_0.txt', '_copy.py', '_collections.py', '_
↪itertools.py', '_math.py', 'TextFile_3.txt']
```

If we want to find only files and not folders/directories, we can do following list comprehension.

```
print([f for f in os.listdir(path_to_look) if os.path.isfile(f)])
```

```
['TextFile_1.txt', 'CsvFile_0.csv', 'readme.txt', 'TextFile_4.txt', '_os.py', 'CsvFile_4.
↪csv', '_types_typing.py', 'CsvFile_1.csv', '_csv.py', 'CsvFile_2.csv', '_warnings.py',
↪'TextFile_2.txt', '_random_module.py', '_time.py', '_pathlib.py', '_sys.py', 'CsvFile_
↪3.csv', 'TextFile_0.txt', '_copy.py', '_collections.py', '_itertools.py', '_math.py',
↪'TextFile_3.txt']
```

If we want to find files with a specific extension, we can do as following

```
print([f for f in os.listdir(path_to_look) if os.path.isfile(f) and f.endswith(".txt")])
```

```
['TextFile_1.txt', 'readme.txt', 'TextFile_4.txt', 'TextFile_2.txt', 'TextFile_0.txt',
↪'TextFile_3.txt']
```

If we want to find files with a specific extension, and starting with a specific name, we can do as following

```
print([f for f in os.listdir(path_to_look) if os.path.isfile(f) and f.endswith(".txt")␣
↪and f.startswith('TextFile_')])
```

```
['TextFile_1.txt', 'TextFile_4.txt', 'TextFile_2.txt', 'TextFile_0.txt', 'TextFile_3.txt
↪']
```

number of lessons in python-seekho book

```
scripts = [fname  for f in os.walk(python_seekho_scripts) for fname in f[2] if fname.
↪endswith('.py')]
print(scripts)
```

```
['stack_vs_concatenate.py', 'dimensions_axis.py', 'digitize.py', 'nested_functions.py',
↪'for_loops.py', 'installing_packages.py', 'conditional_statements.py', 'copying_lists.
↪py', 'args_and_kwargs.py', 'exceptions.py', 'lists.py', 'while_loops.py',
↪'dictionaries.py', 'generators.py', 'functions.py', 'iterator_vs_iterable.py',
↪'variables.py', 'operators.py', 'sets.py', 'io_operations.py', 'global_vs_local.py',
↪'sequential_data.py', 'keywords.py', 'print_function.py', 'reg_expressions.py',
↪'interfacing_with_fortran.py', 'parallel_processing.py', 'interfacing_with_c_cpp.py',
↪'unit_testing.py', 'interfacing_with_matlab.py', 'interfacing_with_cpp.py', 'speeding_
↪up_with_numba.py', 'scripts_to_executables.py', '_cython.py', 'class.py', '__getattr__.
↪py', 'init.py', 'inheritance.py', 'call.py', 'class_vs_instance_attributes.py',
↪'attributes.py', 'str_repr.py', 'class_methods.py', 'introduction.py', 'public_vs_
↪private_attributes.py', 'magical_methods.py', 'methods.py', 'static_methods.py',
↪'getattr_setattr.py', 'descriptors.py', 'property.py', 'speeding_up.py', 'apply.py',
↪'pivot_vs_melt.py', 'multi-indexing.py', 'groupby.py', 'working_with_timeseries.py',
↪'dataframe_vs_series.py', 'io.py', 'indexing_vs_slicing.py', '_os.py', '_types_typing.
↪py', '_csv.py', '_warnings.py', '_random_module.py', '_time.py', '_pathlib.py', '_sys.
↪py', '_copy.py', '_collections.py', '_itertools.py', '_math.py', '_bar.py', '_plot.py',
↪ '_contour.py', 'marginal_plots.py', '_scatter.py', '_circular_bar.py', '_parallel_
↪plot.py', 'intro.py', '_hist.py', '_dumbell.py', '_regplot.py', '_lollipop.py', '_
↪ridge.py', '_boxplot.py', 'pie.py', '_taylor.py', '_violin.py', '_imshow.py', '_spider.
↪py']
```

```
print(f"Total number of lessons in python-seekho book are {len(scripts)}.")
```

```
Total number of lessons in python-seekho book are 91.
```

Total running time of the script: (0 minutes 0.012 seconds)

2.2 sys

Important: This lesson is still under development.

```
import sys
```

path

argv

stdout

stderr

version

```
print(sys.version)
```

```
3.9.19 (main, Jun 18 2024, 09:35:09)
[GCC 11.4.0]
```

prefix

executable

```
print(sys.executable)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/envs/latest/bin/python
```

stdin

modules

getsizeof()

getrefcount()

maxsize

platform

Total running time of the script: (0 minutes 0.001 seconds)

2.3 time and dateandtime

Important: This lesson is still under development.

time

```
import time
```

time()

```
time.time()
```

```
1731353539.9699662
```

```
start = time.time()

# do something

print(time.time() - start, "seconds")
```

```
7.2479248046875e-05 seconds
```

sleep()

```
for i in range(5):
    time.sleep(0.5)
    print(i, end = " ")
```

```
0 1 2 3 4
```

asctime()

```
from time import asctime
asctime()
```

```
'Mon Nov 11 19:32:22 2024'
```

clock()

daylight

localtime()

```
print(time.localtime())
```

```
time.struct_time(tm_year=2024, tm_mon=11, tm_mday=11, tm_hour=19, tm_min=32, tm_sec=22,␣
↪tm_wday=0, tm_yday=316, tm_isdst=0)
```

timezone

ctime()

```
print(time.ctime())
```

```
Mon Nov 11 19:32:22 2024
```

datetime

```
from datetime import datetime
```

datetime.now()

```
now = datetime.now()

print(now)
```

```
2024-11-11 19:32:22.475105
```

```
print(type(now))
```

```
<class 'datetime.datetime'>
```

datetime.ctime()

```
datetime.ctime(now)
```

```
'Mon Nov 11 19:32:22 2024'
```

datetime.strftime()

```
print(datetime.strftime(now, "%Y-%m-%d %H:%M:%S"))
```

```
2024-11-11 19:32:22
```

datetime.strptime()

datetime.isoformat()

```
datetime.isoformat(now)
```

```
'2024-11-11T19:32:22.475105'
```

```
datetime.isoformat(now, sep=" ")
```

```
'2024-11-11 19:32:22.475105'
```

datetime.fromisoformat()

datetime.toordinal()

datetime.fromtimestamp()

timedelta

```
from datetime import timedelta

current = datetime.now()

current + timedelta(seconds=10)
```

```
datetime.datetime(2024, 11, 11, 19, 32, 32, 476729)
```

```
current + timedelta(days=10)
```

```
datetime.datetime(2024, 11, 21, 19, 32, 22, 476729)
```

```
current + timedelta(weeks=10)
```

```
datetime.datetime(2025, 1, 20, 19, 32, 22, 476729)
```

```
current - timedelta(hours=10)
```

```
datetime.datetime(2024, 11, 11, 9, 32, 22, 476729)
```

We can compare two datetime objects

```
past = current - timedelta(hours=10)

current > past
```

```
True
```

```
current < past
```

```
False
```

```
future = current + timedelta(hours=10)

current < future
```

```
True
```

```
current > future
```

```
False
```

strftime()

```
now.strftime("%Y%m%d_%H%M%S")
```

```
'20241111_193222'
```

```
now.strftime("%d %B %Y %H:%M:%S")
```

```
'11 November 2024 19:32:22'
```

Total running time of the script: (0 minutes 2.509 seconds)

2.4 copy

Important: This lesson is still under development.

```
import copy
```

copy()

deepcopy()

Total running time of the script: (0 minutes 0.000 seconds)

2.5 collections

Important: This lesson is still under development.

```
books = {"AlSadr": ["Our Philosophy", "Our Economy"],
         "Mutahri": ["Divine Justice", "Man and Destiny"]}
```

defaultdict

In a normal dictionary, when we try to access value of a non-existent key, we will get `KeyError`.

```
man = {'name': 'Ali'}

print(man['name'])
```

```
Ali
```

```
# uncomment following line
# man['height'] # KeyError
```

If however, we want to avoid this error, we can make use of `defaultdict` function. The input to `defaultdict` must be a callable.

```
from collections import defaultdict

print(int())
```

```
0
```

```
man = defaultdict(int)
man['weight'] = 75
print(man['height'])
```

```
0
```

deque

```
from collections import  deque
a = deque([1,2,3], maxlen=5)

print(a)
```

```
deque([1, 2, 3], maxlen=5)
```

```
a.append(4)
print(a)
```

(continues on next page)

(continued from previous page)

```
a.append(5)
print(a)

a.append(6)
print(a)
```

```
deque([1, 2, 3, 4], maxlen=5)
deque([1, 2, 3, 4, 5], maxlen=5)
deque([2, 3, 4, 5, 6], maxlen=5)
```

```
a.appendleft(1)
print(a)
```

```
deque([1, 2, 3, 4, 5], maxlen=5)
```

```
my_deque = deque([1, 2, 3], maxlen=3)

my_deque.append(4)

print(my_deque)
```

```
deque([2, 3, 4], maxlen=3)
```

namedtuple

```
from collections import  namedtuple
```

OrderedDict

```
from collections import  OrderedDict
```

Counter

```
from collections import  Counter
```

KeysView

It is used to check whether an object is the keys of a dictionary or not.

```
try:
    from collections import KeysView
except ImportError:  # python>3.9
    from collections.abc import KeysView
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/_collections.py:91: DeprecationWarning: Using or importing the ABCs␣
↪from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3,␣
↪and in 3.10 it will stop working
  from collections import KeysView
```

if we wish to check that whether a python object is key or value, we can achieve this as following

```
isinstance(books.keys(), KeysView)  # -> True
```

```
True
```

ValuesView

It is used to check whether an object is the values of a dictionary or not.

```
try:
    from collections import ValuesView
except ImportError:  # python>3.9
    from collections.abc import ValuesView
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/_collections.py:107: DeprecationWarning: Using or importing the ABCs␣
↪from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3,␣
↪and in 3.10 it will stop working
  from collections import ValuesView
```

similarly for dictionary values

```
isinstance(books.values(), ValuesView)  # -> True
```

```
True
```

Reversible

```
try:
    from collections import Reversible
except ImportError:  # python>3.9
    from collections.abc import Reversible
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/_collections.py:117: DeprecationWarning: Using or importing the ABCs␣
↪from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3,␣
↪and in 3.10 it will stop working
  from collections import Reversible
```

Set

```
try:
    from collections import Set
except ImportError:  # python>3.9
    from collections.abc import Set
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/_collections.py:126: DeprecationWarning: Using or importing the ABCs␣
↪from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3,␣
↪and in 3.10 it will stop working
  from collections import Set
```

Sequence

```
try:
    from collections import Sequence
except ImportError:  # python>3.9
    from collections.abc import Sequence
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules/_collections.py:135: DeprecationWarning: Using or importing the ABCs␣
↪from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3,␣
↪and in 3.10 it will stop working
  from collections import Sequence
```

Total running time of the script: (0 minutes 0.006 seconds)

2.6 types and typing

Important: This lesson is still under development.

```
from typing import Any
from typing import  Callable
from typing import  Optional, Generator
from typing import  NamedTuple
from typing import  Iterator


from types import LambdaType
from types import FunctionType
from types import MethodType
from types import GeneratorType
```

Total running time of the script: (0 minutes 0.000 seconds)

2.7 math

Important: This lesson is still under development.

```
import math

# on float

# on sequence (list, array, tuple) with multiple operations
```

Total running time of the script: (0 minutes 0.000 seconds)

2.8 pathlib

Important: This lesson is still under development.

```
import os
from pathlib import Path
```

Path

```
path = Path(os.getcwd())
print(path)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪builtin_modules
```

```
print(type(path))
```

```
<class 'pathlib.PosixPath'>
```

Total running time of the script: (0 minutes 0.001 seconds)

3.9 warnings

Important: This lesson is still under development.

```
import warnings
```

warn()

catch_warnings

filterwarnings()

simplefilter()

showwarning()

Total running time of the script: (0 minutes 0.000 seconds)

2.10 random

Important: This lesson is still under development.

```
import random
```

random

We can use random function to generate a random number between 0 and 1.

```
print(random.random())
```

```
0.8711341291444393
```

every time we call `random.random()`, the result will be different and therefore unpredictable

```
print(random.random())
```

```
0.49967722025114714
```

```
print(random.random())
```

```
0.5382664759363507
```

randint

If we want to generate a random integer between two ranges, we can make use of randint.

```
print(random.randint(1, 10))
```

```
10
```

choice()

If we want to select a value randomly from a given sequence, we can make use of random.choice function.

```
random.choice([1,2,3,4])
```

```
4
```

```
random.choice([1,2,3,4])
```

```
4
```

choices()

However, if we want to select more than one value, but randomly, from a sequence, we can then make use of random.choices.

```
random.choices([1,2,3,4,5, 6, 7])
```

```
[2]
```

```
random.choices([1,2,3,4,5], k=2)
```

```
[3, 2]
```

sample()

The sample function is very much similar to `choices` function.

```
random.sample([1,2,3,4,5], k=2)

dakoos = ['generals', 'siyasatdan', 'bureaucracy', 'saiths', 'anchors']
random.sample(dakoos, k=2)
```

```
['siyasatdan', 'generals']
```

seed()

Suppose someone is making use of `random.random()` function in his script and getting some results. If you use the same script, your result will be different because we have seen that `random()` function generates a different number upon every call.

How can we make use of random() function and still get reproducible results?

The seed function from random module is used to generate reproducible results. Let's generate five random numbers using random.random() function inside a for loop.

```
for i in range(5):
  print(random.random())
```

```
0.0443122651819966
0.058113746852377135
0.08920798131768548
0.575634928119022
0.31067137246830345
```

The above five numbers are *completely* random in the sense that we can not predict, what would be the next number. If we call the random.random() again, the newly generated random numbers will again be random and unpredictable.

```
for i in range(5):
  print(random.random())
```

```
0.6259167207212634
0.6643058189256204
0.0778389936529621
0.9230876931758452
0.4085553072679219
```

Now, Let's set the random seed and then generate the random numbers using random.random()

```
random.seed(313)

for i in range(5):
  print(random.random())
```

```
0.5646032330638283
0.9693332662214504
```

(continues on next page)

(continued from previous page)

```
0.42674278078547345
0.8722737879866462
0.19282451151232616
```

We have generated 5 random numbers so far after setting the ranomd seed with 313. If we reset the random seed again, the next five random numbers that will be generated will be exactly same as the first five random numbers that were generated (above) when we had set the seed to 313 initially.

```
random.seed(313)

for i in range(5):
  print(random.random())
```

```
0.5646032330638283
0.9693332662214504
0.42674278078547345
0.8722737879866462
0.19282451151232616
```

This means, every time we set the random seed, the generated numbers will be random but reproducible.

Question: (Can you) Predict the random numbers generated as a result of following code?

```
# random.seed(92)
# for i in range(5):
#   print(random.random())
```

Consider for example a function which uses random numbers in it using `random.random()`. Now every time we run the function, the results will be different. This means the output of our function will not be reproducible. We can set the random seed either globally (at the start of script) or locally (inside the function) in order to make our results reproducible.

Question: What will be the last value printed by the following code?

```
# random.seed(313)
# for i in range(7):
#   print(random.random())
```

Setting the random seed does not only affect random.random() function, but if affects all functions of random module. Consider for example random.choice function.

```
for _ in range(3):
  print(random.choice(dakoos))
```

```
siyasatdan
generals
bureaucracy
```

```
random.seed(313)

for _ in range(3):
  print(random.choice(dakoos))
```

```
anchors
siyasatdan
saiths
```

```
random.seed(313)

for _ in range(3):
  print(random.choice(dakoos))
```

```
anchors
siyasatdan
saiths
```

shuffle()

```
print(dakoos)

random.shuffle(dakoos)

print(dakoos)
```

```
['generals', 'siyasatdan', 'bureaucracy', 'saiths', 'anchors']
['siyasatdan', 'bureaucracy', 'anchors', 'generals', 'saiths']
```

```
random.shuffle(dakoos)

print(dakoos)
```

```
['generals', 'saiths', 'bureaucracy', 'anchors', 'siyasatdan']
```

Question: Write a function called `pseudo_shuffle`, which suffles the *dakoos* list, but the order of values in the returned/shuffled list is always same. This means calling `pseudo_shuffle(dakoos)` multile times return same order in the list. It should be noted that the function `pseudo_shuffle` must make use of `random.suffle` function inside it.

randrange

```
random.randrange(1, 10)
```

```
3
```

```
random.randrange(1, 10, step=2)
```

```
7
```

getstate()

setstate()

distributions

Total running time of the script: (0 minutes 0.007 seconds)

2.11 csv

Important: This lesson is still under development.

```
import csv

# with open('csv_file.csv', 'r') as f:
#     reader = csv.reader(f)
#     data = [row for row in reader]
```

Total running time of the script: (0 minutes 0.000 seconds)

2.12 itertools

Important: This lesson is still under development.

```
import itertools
```

combinations

```
x = [1,2,3]
```

```
print([x for x in itertools.combinations(x, 2)])
```

```
[(1, 2), (1, 3), (2, 3)]
```

```
print([x for x in itertools.combinations(x, 3)])
```

```
[(1, 2, 3)]
```

```
print([x for x in itertools.combinations(x, 10)])
```

```
[]
```

product

```
for prod in (itertools.product([1, 2], [11, 12])):
    print(prod)
```

```
(1, 11)
(1, 12)
(2, 11)
(2, 12)
```

```
for prod in (itertools.product([1,2,3], [11, 12, 13])):
    print(prod)
```

```
(1, 11)
(1, 12)
(1, 13)
(2, 11)
(2, 12)
(2, 13)
(3, 11)
(3, 12)
(3, 13)
```

```
for prod in (itertools.product([1, 2], [11, 12], [21, 23])):
    print(prod)
```

```
(1, 11, 21)
(1, 11, 23)
(1, 12, 21)
(1, 12, 23)
(2, 11, 21)
(2, 11, 23)
(2, 12, 21)
(2, 12, 23)
```

permutations

```
print([x for x in itertools.permutations([1,2,3])])
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

```
print([x for x in itertools.permutations([1,2,3], 2)])
```

```
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

repeat

Repeats a value n times

```
value = 10
n = 4

rep = itertools.repeat(value, n)
print(rep)
```

```
repeat(10, 4)
```

```
print(type(rep))
```

```
<class 'itertools.repeat'>
```

```
for val in rep:
    print(val)
```

```
10
10
10
10
```

Total running time of the script: (0 minutes 0.005 seconds)

2.7.3 3. oop

Tutorials concerning object oriented programming

3.1 introduction

The first thing to understanding in object oriented programming in python is the concept of `type`.

In python everything is an `object` and it has a `type`. The `type` of an `object` tells, what kind of characteristics it has, or what kind of operations it can perform or can be performed on it.

```
a_int = 12
print(type(a_int))
```

```
<class 'int'>
```

```
a_float = 12.0
print(type(a_float))
```

```
<class 'float'>
```

```
a_name = 'Ali'
print(type(a_name))
```

```
<class 'str'>
```

Even the functions have their types

```
def print_name(the_name):
    print('name is: ', the_name)

print(type(print_name))
```

```
<class 'function'>
```

```
import math

print(type(math))
```

```
<class 'module'>
```

```
looters = ['zardari', 'nawaz', 'establishment']
print(type(looters))
```

```
<class 'list'>
```

```
insan = {'name': 'ali', 'age': 30, 'weight': 72.5}
print(type(insan))
```

```
<class 'dict'>
```

```
for key, val in insan.items():
    print(type(key), type(val))
```

```
<class 'str'> <class 'str'>
<class 'str'> <class 'int'>
<class 'str'> <class 'float'>
```

Since *insan* is an instance of class `dict`, therefore we can access all functions (methods) of `dict` class through its instance *insan*. These methods include `pop` or `update` etc.

```
insan.pop('weight')
```

```
72.5
```

```
print(insan)
```

```
{'name': 'ali', 'age': 30}
```

We can always check the methods and attributes available for the instance of a class by `dir(object)`

```
print(dir(insan))
```

```
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__
↪', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__
↪gt__', '__hash__', '__init__', '__init_subclass__', '__ior__', '__iter__', '__le__', '_
↪_len__', '__lt__', '__ne__', '__new__', '__or__', '__reduce__', '__reduce_ex__', '__
↪repr__', '__reversed__', '__ror__', '__setattr__', '__setitem__', '__sizeof__', '__str_
↪_', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
↪'popitem', 'setdefault', 'update', 'values']
```

Any method/attribute that starts with single or double underscore "_" is not for public use. Considering this, we can use any attribute/method of an *insan* which we printed above. For example, we can do *insan.update* or *insan.vlaues* or *insan.keys* etc. For `methods`, we have to *call* them like insan.values() and for other attributes, we don't need to call them.

```
print(dir(a_name))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
↪'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
↪'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
↪ '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
↪'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
↪'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find
↪', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
↪'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
↪ 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix
↪', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
↪'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate
↪', 'upper', 'zfill']
```

The attributes of *a_name* are different as compared to those of *insan*. Therefore, we can not do *insan.startswith()* or *a_name.values()*.

Question: What methods/functions can be applied on *a_int* and *a_float* objects defined at the start of this lesson? Give examples of five such functions by applying them.

Total running time of the script: (0 minutes 0.004 seconds)

3.2 creating class

A class in python can be considered as a collection of functions which share some data (attributes). However, a class need not to have functions in it i.e. it can only consist of attributes.

We can define a minimal class as follows

```
class Insan:
    pass
```

A `class` definition consists of two parts:

- header: keyword `class` and name of class and then listing of other classes from which this class inherits i.e. superclasses. 3rd argument is optional.
- body: indented statements, Above we have only one statement i.e. `pass`.

Using classes

To use a class, we first have to initiate or instantiate it. This is done by calling the class as if it were a function. For example, we can create an object of `Insan` class as follows

```
ali = Insan()
husain = Insan()
```

Above we have created two objects which are called instances of a class. *ali* and *husain* are instances of *Insan* class. If we check the type of these instances we can confirm it.

```
type(ali), type(husain)
```

```
(<class '__main__.Insan'>, <class '__main__.Insan'>)
```

```
shabir = husain
```

```
print(husain == shabir)
```

```
True
```

```
print(husain == ali)
```

```
False
```

So *husain* and *ali* are different although they are instances of same class. They are different because they are different instances of the *Insan* class.

Total running time of the script: (0 minutes 0.002 seconds)

3.3 Attributes

Attribute usually means some specific quality. In python attribute is different than property.

```
class Insan:
    pass


x = Insan()
y = Insan()
```

We have created two objects/instances of class *Insan* namely *x* and *y*. We can bind attributes to class instances as follows:

```
x.name = "Ali"
x.dob = "600"
```

```
y.name = "Hasan"
y.dob = "625"
```

We can now check that the attributes have been associated with *x* and *y*.

```
print(x.name)
```

```
Ali
```

```
print(y.dob)
```

```
625
```

It should be noticed that we have associated *name* and *dob* attributes to instances of *Insan* class i.e., *x* and *y* and not with *Insan* class. This is a dynamic way of attribute creation. Usually attributes are built inside the class, which we will cover later. what attributes does the instance *x* has? We can find it out as following

```
print(x.__dict__)
```

```
{'name': 'Ali', 'dob': '600'}
```

```
print(y.__dict__)
```

```
{'name': 'Hasan', 'dob': '625'}
```

We can also bind attributes to class names as well.

```
class Insan(object):
    pass


x = Insan()

Insan.cast = "Jat"
```

attribute *cast* is currently associated with instance *x*.

```
print(x.cast)
```

```
Jat
```

We can change the attribute *cast* ` associated with instance *x* as below

```
x.cast = "cheema"
```

```
print(x.cast)
```

```
cheema
```

what is attribute *cast* associated with class name 'Insan'?

```
print(Insan.cast)  # >> Jat
```

```
Jat
```

```
y = Insan()
print(y.cast)  # what is attribute `cast` associated with instance `y`?
```

```
Jat
```

y was never assigned an attributed named *cast*. Still it threw a value, why?

Let's make the attribute *cast* associated with *Insan* as *Insaniyat* now

```
Insan.cast = "insaniyat"
print('y_cast: ', y.cast)
print('x_cast: ', x.cast)
```

```
y_cast:  insaniyat
x_cast:  cheema
```

```
print(x.__dict__)
```

```
{'cast': 'cheema'}
```

```
print(y.__dict__)
```

```
{}
```

empty so if we call the attribute *cast* for instance *y*, python will first look into y attributes and if it does not find then it will look into attributes of *Insan*

```
# mappingproxy({'__module__': '__main__', '__weakref__': , '__doc__': None, '__dict__': , 'cast
↪': 'insaniyat'})
print(Insan.__dict__)
```

```
{'__module__': '__main__', '__dict__': <attribute '__dict__' of 'Insan' objects>, '__
↪weakref__': <attribute '__weakref__' of 'Insan' objects>, '__doc__': None, 'cast':
↪'insaniyat'}
```

So even though *y* instance itself does not have an attribute named *cast* so it checked whether the attribute *cast* exists in attributes of class *Insan*? If yes (which is the case) so *y* gets the attribute from *Insan* while *x* already had attribute named *cast* so it did not get attribute from class *Insan*.

If we try to get an attribute which is non-existing, we will get an `AttributeError`

```
# uncomment following line
# x.age  # >> AttributeError: `Insan` object has no attribute `age`
```

One way to prevent such error is to provide a default value for the attribute by

```
getattr(x, 'age', 90)
```

```
90
```

We can bind attributes to function names similarly

```
def chor(name):
    return name + ' chor hai'


chor.age = 61
print(chor.age)
```

```
61
```

We can use this trick to count number of function calls

```
def chor(name):
    chor.no_of_calls = getattr(chor, "no_of_calls", 0) + 1
    return name


for i in range(10):
    chor('nawaz')
```

```
print(chor.no_of_calls)
```

```
10
```

To properly create class instances we need to define *methods* in the class body, which we will learn next

Total running time of the script: (0 minutes 0.006 seconds)

3.4 methods

This lesson discusses the concept of method in python.

Forget for the time being what the word *methods* mean in English. The term *method* here means *functions associated with classes* Let's write a simple function which takes an *object* as input and prints the *name* attribute of that object"

```
def say_salam(obj):
    print("Salam, I am " + obj.name + "!")
    return
```

Let's also define a simple class named *Insan* as we did in previous examples.

```
class Insan:
    pass
```

We can use the function *salam* by using the instance of class *Insan* which is *x*.

```
x = Insan()
x.name = "Ali"
say_salam(x)  # >> Salam, I am Ali!
```

```
Salam, I am Ali!
```

`say_salam` is a function at this time, and we can verify this by checking its type

```
type(say_salam)
```

but we can bind/link it to class *Insan* as following

```
def say_salam(obj):
    print("Salam, I am " + obj.name)


class Insan:
    taruf = say_salam
```

Now we can make use of the function `say_salam` which is linked to class `Insan` as following

```
x = Insan()

x.name = "Ali"

Insan.taruf(x)
```

```
Salam, I am Ali
```

attributes in the `Insan` class are:

```
print(Insan.__dict__)
```

```
{'__module__': '__main__', 'taruf': <function say_salam at 0x7f1369535ee0>, '__dict__':
↪<attribute '__dict__' of 'Insan' objects>, '__weakref__': <attribute '__weakref__' of
↪'Insan' objects>, '__doc__': None}
```

taruf is a method and can be called as

```
x.taruf()
```

```
Salam, I am Ali
```

so *Insan.taruf* and *x.taruf* are equivalent. Although the method *say_salam* takes one argument `obj` as input but we did not provide any input argument while calling it through `x.taruf()` and no error was thrown? This is because when the function `say_salam` was linked to the class `Insan`, and when we call this method, the first argument is by default the instance i.e. `x` in this case. In this case we defined the method outside the body of class `Insan` and then linked it to class, but this is not the usual way to define methods. The proper way is to define it inside the class(indented) We connect this function which is method now to the class by its first argument which is usually `self`. `self` corresponds to the `Insan` object `x`

```
class Insan:
    def say_salam(self):
        print("Salam, I am " + self.name)
        return


ali = Insan()
```

uncomment following line ali.say_salam() # AttributeError

The problem with above code is that the class `Insan` does not have an attribute `name`.So the class must have an attribute `name` before using the method `say_salam`.

```
print(ali.__dict__)
```

```
{}
```

Lets define the attribute before using the method `say_salam`

```
class Insan:
    def say_salam(self):
        print("Salam, I am " + self.name)


ali = Insan()
ali.name = 'Ali'  # define an attribute of instance ali
ali.say_salam()
```

```
Salam, I am Ali
```

```
print(ali.__dict__)
```

```
{'name': 'Ali'}
```

so whats the difference btw `method` and `function`? `self` is just a convention, we can use `this`, `apna` or any other keyword but it is better to just follow the convention so that others can follow your work

```
class Insan:

    def say_salam(apna):
        print("Salam, I am " + apna.name)


ali = Insan()
ali.name = 'Ali'  # define an attribute of instance ali
ali.say_salam()
```

```
Salam, I am Ali
```

As said earlier, defining class attributes outside class is not proper way. We need the class `Insan` to have the attribute `name` before using the method `say_salam`. So we need a systematic way that the class, upon its creation (initialization) must have the attribute `name`.This is done with `init` method, which we will learn in next lesson.

Total running time of the script: (0 minutes 0.004 seconds)

3.5 init method

This method is used to define the attributes of a class which we want that class to have right upon its creation. This means the code in `init` method is executed at the time of creation of an instance of class. Thus we use `init` method to initialize the class, hence name `init`. It can be anywhere inside class but convention is to put it at the top inside a class.

```
class Insan:
    def __init__(self):
        self.legs = '2 legs'
        print("__init__ method inside class is being executed!")
```

```
x = Insan()
x.legs
```

```
__init__ method inside class is being executed!

'2 legs'
```

we see that although we only created an instance of class (we call it initialization of an instance as well) and the code inside init method (we don't use the word function for it anymore) got executed. The proper way to add *taruf* method to our Insan class is:

```
class Insan:

    def __init__(self, name=None, legs='2 legs'):
        self.name = name
        self.legs = legs

    def taruf(self):
        if self.name:
            print("Salam, I am " + self.name + ' and I have ' + self.legs)
        else:
            print("Salam, I am a Insan but my name has not yed been defined" + ' and I␣
↪have ' + self.legs)
```

```
x = Insan()
x.taruf()
```

```
Salam, I am a Insan but my name has not yed been defined and I have 2 legs
```

```
y = Insan("Ali", legs='1 leg')
y.taruf()
```

```
Salam, I am Ali and I have 1 leg
```

Notice how we passed the string `Ali` when we were creating class instance and got it transferred to the method `taruf` without being explicitly doing it. The method `taruf` takes no input arguments (actually it takes one, which it does implicitly) however it prints the value of *name* which we provided to it when we were creating the class instance `y`.

Let's create a new class instance and use a different name

```
y = Insan("Hasan")
y.taruf()  # >> Salam, I am Hasan
```

```
Salam, I am Hasan and I have 2 legs
```

The attribute `name` is shared by the whole class and can be accessed anywhere in the class by using the word `self` at its start. This is because we initialized it inside `init` method.

```
print(y.name)
```

```
Hasan
```

```
print(y.legs)
```

```
2 legs
```

```
print(y.__dict__)
```

```
{'name': 'Hasan', 'legs': '2 legs'}
```

Total running time of the script: (0 minutes 0.003 seconds)

3.6 str and repr methods

Default behaviour

Unless we overwrite the default method, for most classes the default output does not mean something useful.

```
class Insan(object):
    pass


ali = Insan

print(str(ali))
repr(ali)
```

```
<class '__main__.Insan'>

"<class '__main__.Insan'>"
```

If a class does not have explicit definition of `str` or `repr` methods, python will used the default output.

```
name = 4

str(name)
```

```
'4'
```

```
repr(name)
```

```
'4'
```

name belongs to class `int` and this class has ``str` and *repr* methods so python did not give default output rather used the methods from *int* class. but until here we see no apparent difference.

Overwriting

If we override one of the two methods:

```
class Insan(object):

    def __str__(object):
        return "Insan class"


ali = Insan()
str(ali)
```

```
'Insan class'
```

```
repr(ali)
```

```
'<__main__.Insan object at 0x7f13695e36a0>'
```

```
class Insan(object):

    def __repr__(object):
        return "Insan class"


ali = Insan()
str(ali)
```

```
'Insan class'
```

```
repr(ali)
```

```
'Insan class'
```

So if we overwrite `repr` method, `str` is also overwritten (second case) but if we only overwrite `str` method, `repr` will not be overwritten unless we do it explicitly (first case). `__str__` is readable and `__repr__` is unambiguous

Usually the purpose of *str* method to to give a readable output while that of *repr* is give an unambiguous output. consider the following example.

```
name = 'ali'

repr(name)
```

```
"'ali'"
```

```
name2 = eval(repr(name))

print(name2 == name)
```

```
True
```

```
name3 = eval(str(name))

print(name3 == name)
```

```
False
```

repr is such a representation of an object that by evaluating it (by using eval) it will return back the object. type(name) and type(name2) will be same. This is why name and name2 were same. While *str* gives such a representation of an object which is for better reading purpose.

```
type(name), type(name2)
```

```
(<class 'str'>, <class 'str'>)
```

```
type(name3)
```

```
class Revolution:

    def __init__(self, name, dob):
        self.name = name
        self.dob = dob

    def __repr__(self):
        return "Revolution('" + self.name + "', " + str(self.dob) + ")"

    # def __str__(self):
    #     return "Revolution('" + self.name + "', " +  str(self.dob) +  ")"


x = Revolution("Eslami", 1979)

x_str = str(x)
print(x_str)
print("Type of x_str: ", type(x_str))
new = eval(x_str)
print(new)
print("Type of new:", type(new))
```

```
Revolution('Eslami', 1979)
Type of x_str:  <class 'str'>
Revolution('Eslami', 1979)
Type of new: <class '__main__.Revolution'>
```

So we only overwrote `repr`, but `str` also gave same output as that of `repr`. Moreover check the types of `str` and `repr`.

In above case we were able to evaluate *str* representation of *name* but in some cases even this may not be possible as in following case:

```
class Revolution:

    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year

    def __repr__(self):
        return "Revolution('" + self.name + "', " + str(self.build_year) + ")"

    def __str__(self):
        return "Name: " + self.name + ", took place in Year: " + str(self.build_year)


x = Revolution("Eslami", 1979)

x_str = str(x)
print(x_str)
print("Type of x_str: ", type(x_str))
```

```
Name: Eslami, took place in Year: 1979
Type of x_str:  <class 'str'>
```

```
# uncomment following line
# new = eval(x_str)  # SyntaxError
```

```
x_repr = repr(x)
new = eval(x_repr)
print(new)
```

```
Name: Eslami, took place in Year: 1979
```

Consider another example where `str` output can not be evaluated

```
import datetime

aaj = datetime.datetime.now()
aaj_str = str(aaj)
print(aaj_str)
```

```
2024-11-11 19:32:23.199371
```

```
aaj_repr = repr(aaj)
print(aaj_repr)
```

```
datetime.datetime(2024, 11, 11, 19, 32, 23, 199371)
```

```
# uncomment following line
# eval(aaj_str)  # SyntaxError
```

```
eval(aaj_repr)
```

```
datetime.datetime(2024, 11, 11, 19, 32, 23, 199371)
```

So `repr` is a pure pythonic representation of an object which can be evaluated while `str` is for reading purpose.

Total running time of the script: (0 minutes 0.006 seconds)

3.7 public vs private attributes

Based on accessibility, attributes can be categorized into 3 categories: * Public * Protected * Private

** Naming ** | ** Type ** | ** Description ** |
— | — | — |
name | public | The attribute is available both inside and outside class |

|_name | protected | should not be should outside class definition | |__name | private | inaccessible and invisible outside class definition |

```
class Insan():

    def __init__(self):
        self.__ghar = "I am private"
        self._gari = "I am protected"
        self.name = "I am public"


x = Insan()

print(x.name)
```

```
I am public
```

```
print(x._gari)
```

```
I am protected
```

```
# uncomment following line
# x.__ghar  # AttributeError
```

Although the `Insan` class do have an attribute anmed `__priv`, however the error is saying **Insan object has no attribute __priv**. What better privacy can there be? We can also set the values of public attributes.

```
x.name = 'hasan'

print(x.name)
```

```
hasan
```

As said earlier private attribute is not accessible outside the class but we can use them inside class definition as follows:

```
class Insan():

    def __init__(self):
        self.__ghar = "I am private"
        self._gari = "I am protected"
        self.name = "I am public"

    def get_ghar(self):
        return self.__ghar


ali = Insan()
ali.get_ghar()
```

```
'I am private'
```

name mangling

Every attribute with double underscore will be changed to `object._class__attribte`

```
print(ali._Insan__ghar)
```

```
I am private
```

Ways of accessing and setting attribute values

```
class Insan:

    def __init__(self, name=None, dob=2000):
        self.__name = name
        self.__dob = dob

    def say_salam(self):
        if self.__name:
            print("Salam, I am " + self.__name)
        else:
            print("Salam, I am a person without a name")

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_dob(self, by):
        self.__dob = by
```

(continues on next page)

(continued from previous page)

```
    def get_dob(self):
        return self.__dob

    def __repr__(self):
        return "Insan('" + self.__name + "', " + str(self.__dob) + ")"

    def __str__(self):
        return "Name: " + self.__name + ", born in Year: " + str(self.__dob)


x = Insan("Mutahri", 1920)
y = Insan("Sadr", 1935)

for fard in [x, y]:
    fard.say_salam()
    if fard.get_name() == "Sadr":
        fard.set_dob(1353)
    print("I was born in the year " + str(fard.get_dob()) + "!")
```

```
Salam, I am Mutahri
I was born in the year 1920!
Salam, I am Sadr
I was born in the year 1353!
```

Total running time of the script: (0 minutes 0.003 seconds)

3.8 class vs instance attributes

```
class Insan:
    ethnicity = "Balochi"


x = Insan()
y = Insan()

print(x.ethnicity)
```

```
Balochi
```

```
print(y.ethnicity)
```

```
Balochi
```

```
print(Insan.ethnicity)
```

```
Balochi
```

```
x.ethnicity = "muhajir"
y.ethnicity
```

```
'Balochi'
```

```
Insan.ethnicity
```

```
'Balochi'
```

```
Insan.ethnicity = "Insaniyat"
Insan.ethnicity
```

```
'Insaniyat'
```

```
y.ethnicity
```

```
'Insaniyat'
```

```
x.ethnicity
```

```
'muhajir'
```

Because the attribute was changed for class and instance `x` was created before this change so `x` still has old attribute value.

```
x.__dict__
```

```
{'ethnicity': 'muhajir'}
```

```
print(y.__dict__)
```

```
{}
```

Because *y* itself does not have attribute `ethnicity` so it looked for the attribute value in class *Insan* and used that value for itself.

```
print(Insan.__dict__)
```

```
{'__module__': '__main__', 'ethnicity': 'Insaniyat', '__dict__': <attribute '__dict__'␣
↪of 'Insan' objects>, '__weakref__': <attribute '__weakref__' of 'Insan' objects>, '__
↪doc__': None}
```

So class attributes and instance attributes can have different values. We can find out attributes of `class` of an `instance` from the *instance* using following code

```
print(x.__class__.__dict__)
```

```
{'__module__': '__main__', 'ethnicity': 'Insaniyat', '__dict__': <attribute '__dict__'␣
↪of 'Insan' objects>, '__weakref__': <attribute '__weakref__' of 'Insan' objects>, '__
↪doc__': None}
```

```
class philosopher:
    Quotes = (
        """What cannot be imagined cannot even be talked about.""",
        """Philosophy is a battle against the bewitchment of our intelligence by means␣
↪of language.,""",
        """The limits of my language means the limits of my world."""
    )

    def __init__(self, name='Wittgenstein', dob=1889):
        self.name = name
        self.build_year = build_year

    # other methods as usual
```

```
for number, quote in enumerate(philosopher.Quotes):
    print(str(number + 1) + ":\n" + quote)
```

```
1:
What cannot be imagined cannot even be talked about.
2:
Philosophy is a battle against the bewitchment of our intelligence by means of language.,
3:
The limits of my language means the limits of my world.
```

Counting instances of a class

```
class Insan:
    counter = 0

    def __init__(self):
        # every time a new instance is created the attribute 'counter' of 'Insan' increases␣
↪by 1
        type(self).counter += 1

    def __del__(self):
        # every time an instance of `Insan` is deleted, the attribute 'counter' of 'Insan'␣
↪decreases by 1
        type(self).counter -= 1


x = Insan()
print("Population count is: : " + str(Insan.counter))
y = Insan()
print("Population count is: : " + str(Insan.counter))
del x
print("Population count is: : " + str(Insan.counter))
del y
print("Population count is: : " + str(Insan.counter))
```

```
Population count is: : 1
Population count is: : 2
Population count is: : 1
Population count is: : 0
```

`type(self)` is evaluated back to *Insan*

Total running time of the script: (0 minutes 0.005 seconds)

3.9 static methods

In previous section we wrote a code to count number of instances of class using a public attribute of a class. If we make this attribute *private* , we can create a method inside the class to acquire its value. In following example, we do it by `PopulationCount` method.

```
class Insan:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def PopulationCount(self):
        return Insan.__counter


ali = Insan()
print(ali.PopulationCount())
hasan = Insan()
print(hasan.PopulationCount())
```

```
1
2
```

So far so good. But we have a problem, first, the `PopulationCount` does not have anything to do with the instance of class *Insan* i.e. *ali* and second, if we want access this method directly from class' instance, we will encounter an error as shown below.

```
# uncomment following line
# Insan.PopulationCount()  # TypeError
```

This is because, the method `PopulationCount` takes one input argument `self`. When we call this method using instance of class, python implicit puts the class' instance, `ali` in this case, as an input argument but now we are not calling this method from instance, so python does not provide the implicit first input argument to this method. We could however, avoid this error by explicitly providing the instance `ali` as first input argument.

```
Insan.PopulationCount(ali)
```

```
2
```

However, this is not a good way. An alternative would be to avoid the `self` statement when defining the method in the class.

```
class Insan:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    def PopulationCount():
        return Insan.__counter


Insan.PopulationCount()
```

```
0
```

But now we can't access this method from the instance, because when we access this method from instance, python implicitly gives instance *ali* as first input argument to this method but the method does not take any input argument as the error message also depicts this.

```
# uncomment following two lines
# ali = Insan()  # TypeError
# ali.PopulationCount()
```

The way to solve this problem is to put the decorate *@staticmethod*. By doing so, python will not put the instance implicitly as first input argument.

```
class Insan:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def PopulationCount():
        return Insan.__counter


print(Insan.PopulationCount())
ali = Insan()
print(ali.PopulationCount())
hasan = Insan()
print(hasan.PopulationCount())
print(Insan.PopulationCount())
```

```
0
1
2
2
```

why to use them?

Static methods are used to group a utility function with a class.

```
class Dates:
    def __init__(self, date):
        self.date = date

    def getDate(self):
        return self.date

    @staticmethod
    def toDashDate(date):
        return date.replace("/", "-")


date = Dates("15-12-2016")
dateFromDB = "15/12/2016"
dateWithDash = Dates.toDashDate(dateFromDB)
```

since `toDashDate` works only for dates, it's logical to keep it inside the Dates class Consider another example. Suppose we have a large number of examples which perform mathematical functions such as `ceil`, `multiply`, `exponent`, `divide` on some input arguments. Then it would be logical to just group all such functions one one class such as `math` and use the functions as `math.ceil(2.2)` or `math.exp(2.3)` etc.

```
class math():

    @staticmethod
    def ceil(x):
        # perform ceil operation
        return

    @staticmethod
    def exp(x):
        # perform exponent operation
        return

    # more methods
```

This increases code readability.

Total running time of the script: (0 minutes 0.003 seconds)

3.10 class methods

```
import datetime
```

In previous section we saw that we can make a method linked to class by removing the keyword `self` from its input arguments and in this way we can call this method from class such as `ClassName.MethodName()`.

```
class Insan:
    __counter = 0
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        type(self).__counter += 1

    def PopulationCount():
        return Insan.__counter


Insan.PopulationCount()
```

```
0
```

Let's say, we also want to count male population, then we can add another static method as follows:

```
class Insan:
    __counter = 0
    __MaleCounter = 0

    def __init__(self, gender):
        if gender == 'male':
            type(self).__MaleCounter += 1

        type(self).__counter += 1

    @staticmethod
    def PopulationCount():
        return Insan.__counter

    @staticmethod
    def MaleCount():
        return Insan.__MaleCounter


ali = Insan('male')
fatima = Insan('female')

Insan.PopulationCount()
```

```
2
```

```
Insan.MaleCount()
```

```
1
```

If we want to now add female counter in above class, one way of doing it is to indirectly calculating it from total counter and male counter as follows

```
class Insan:
    __counter = 0
    __MaleCounter = 0

    def __init__(self, gender):
```

(continues on next page)

(continued from previous page)

```
        if gender == 'male':
            type(self).__MaleCounter += 1

        type(self).__counter += 1

    @staticmethod
    def PopulationCount():
        return Insan.__counter

    @staticmethod
    def MaleCount():
        return Insan.__MaleCounter

    def FemaleCount():
        return Insan.__counter - Insan.__MaleCounter


ali = Insan('male')
fatima = Insan('female')
```

```
print(Insan.PopulationCount())
```

```
2
```

```
print(Insan.MaleCount())
```

```
1
```

```
Insan.FemaleCount()
```

```
1
```

Another way of achieving this is to use the decorator `@classmethod`. The use of this decorator makes sure that when we call this method, the first default implicit argument, that python provides to this method is the class itself. Thus we can make use of some static methods of the class without initializing the class

```
class Insan:
    __counter = 0
    __MaleCounter = 0

    def __init__(self, gender):
        if gender == 'male':
            type(self).__MaleCounter += 1

        type(self).__counter += 1

    @staticmethod
    def PopulationCount():
        return Insan.__counter
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def MaleCount():
        return Insan.__MaleCounter

    @classmethod
    def FemaleCount(cls):
        return cls.__counter - cls.__MaleCounter


ali = Insan('male')
fatima = Insan('female')
zeinab = Insan('female')

print(Insan.PopulationCount())
```

```
3
```

```
print(Insan.MaleCount())
```

```
1
```

```
Insan.FemaleCount()
```

```
2
```

So until here class methods are behaving same as static methods in addition to that we can access other static methods of the class. It may not make a lot of sense but we will come to such use of class method once we cover `inheritance`. Now Consider the following example:

```
class Student(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age


ali = Student('ali', 12)
print(type(ali))
```

```
<class '__main__.Student'>
```

But let's say, for a particular student we don't know his/her age but we know the birth year. Thus we would like to build the class from birth year directly as well.

```
class Student(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def fromBirthYear(cls, name, BirthYear):
        current_year = datetime.date.today().year
        age = current_year - BirthYear
        return cls(name, age)


ali = Student('ali', 12)
print(type(ali))
```

```
<class '__main__.Student'>
```

```
hasan = Student.fromBirthYear('hasan', 1997)
print(type(hasan))
```

```
<class '__main__.Student'>
```

```
print(hasan.age)
```

```
27
```

Thus we used the `class method` to build/initiate the class from birth year. We could have performed the conversion of `BirthYear` to `age` outside the class as well but this way provides a more user friendly interface of the class and the code is more organized.

Total running time of the script: (0 minutes 0.006 seconds)

3.11 property

Let's say we have a for human body metabolism which has temperature as one of its parameters(attributes) and we want to have controle over this parameter in such a way that we can set the temperature from outside as well while the model itself also changes the temperature parameter when it is run.

```
class Model:

    def __init__(self, temp):
        self.temp = temp


x = Model(39)
print(x.temp)

x.temp = 35
print(x.temp)
```

```
39
35
```

Thus we can set the temperature from outside the class. But what if after some time we want to set certain condition on temperature attribute such as the temperature should never be above 45 degrees and below 20 degrees.

```
class Model:

    def __init__(self, temp):
        self.set_temp(temp)

    def set_temp(self, x):
        if x > 45:
            x = 45
        if x < 20:
            x = 20
        self.temp = x

    def get_temp(self):
        return self.temp


x = Model(39)
print(x.temp)
```

```
39
```

We can set the temperature as before but then we may violate the condition as well

```
x.temp = 55
print(x.temp)
```

```
55
```

In order to set the temperature, we have to now make use of method *set_temp*.

```
x.set_temp(55)
print(x.temp)
```

```
45
```

if we want to change the temperature value based upon its current value, we can do as following

```
print(x.get_temp())
x.set_temp(x.get_temp() * 0.9)
print(x.get_temp())
```

```
45
40.5
```

we were able to decrease the temperature but there are two problems.

- **First the statement `x.set_temp(x.get_temp()*0.9)` does not look so elegant,** it would have been much better and clearer if we could do like `x.temp = x.temp*0.9`.
- **There are two ways to set and get temperature which is against the** zen of python[1] which states

There should be one– and preferably only one –obvious way to do it. Based upon above discussion we would have liked `temp` to behave like attribute (*x.temp*) but still be able to perform some checks on it behind the scene (which

[1] https://www.python.org/dev/peps/pep-0020/

we could do by `x.temp()`) and there should be only one way to set and get its value as well. We can solve the second problem by making `temp` a private attribute i.e. by making it `__temp`, and then we have to use only setters `x.set_temp()` and getters `x.get_temp`.

```
class Model:

    def __init__(self, temp):
        self.set_temp(temp)

    def set_temp(self, x):
        if x > 45:
            x = 45
        if x < 20:
            x = 20
        self.__temp = x

    def get_temp(self):
        return self.__temp


x = Model(39)
print(x.get_temp())

x.set_temp(55)
print(x.get_temp())
```

```
39
45
```

But python provides a better solution to solve both of above two problems i.e. the *@property* decorator.

```
class Model:

    def __init__(self, temp):
        self.temp = temp

    @property
    def temp(self):
        # get temperature from the model through some process
        return self._temp

    @temp.setter
    def temp(self, x):
        if x > 45:
            x = 45
        if x < 20:
            x = 20
        self._temp = x
        return


x = Model(39)
print(x.temp)
```

(continues on next page)

(continued from previous page)

```
x.temp = x.temp * 0.9
print(x.temp)
```

```
39
35.1
```

Just a side note, we don't have provide the default value of *temp* upon class initiation. We can make the method to get the current temperature state of the model. In many cases our model will be saved in an external file. For simplicity, suppose, our model is saved as dictionary *MODEL*. We can make use of property to manipulate *temp*.

```
MODEL = {'temp': 39}


class Model:

    def __init__(self):
        pass

    @property
    def temp(self):
        # get temperature from the model through some process
        t = MODEL['temp']
        return t

    @temp.setter
    def temp(self, x):
        if x > 45:
            x = 45
        if x < 20:
            x = 20
        MODEL['temp'] = x
        return


x = Model()
print(x.temp)

x.temp = x.temp * 0.9
print(x.temp)
```

```
39
35.1
```

run the model and check the temperature again

```
MODEL
```

```
{'temp': 35.1}
```

What happens if we skip setter i.e. *@temp.setter*? This will make the attribute readonly, and any user of the class will not be able to modify its value from outside the class using instance.

```
MODEL = {'temp': 39}


class Model:

    def __init__(self):
        pass

    @property
    def temp(self):
        # get temperature from the model through some process
        t = MODEL['temp']
        return t


x = Model()
print(x.temp)
```

```
39
```

```
# uncomment following line
# x.temp = x.temp * 0.9  # AttributeError


print(x.temp)
```

```
39
```

run the model and check the temperature again

Total running time of the script: (0 minutes 0.006 seconds)

3.12 Descriptors

Descriptors are another way to control, what happens when a value of an attribute is set or accessed. We do it while making a class with at least one of `__get__`, `__set__` and `__del__` methods.

```
class MyDescriptor(object):
    """
    Basic descriptor to set and get value.
    """

    def __init__(self, initval=None):
        print("__init__ of MyDescriptor called with initial value: ", initval)
        self.__set__(self, initval)

    def __get__(self, instance, owner):
        print(instance, owner)
        print('Getting  self.val: ', self.val)
        return self.val

    def __set__(self, instance, value):
```

(continues on next page)

(continued from previous page)

```
        print('Setting self.val to ', value)
        self.val = value


class Model(object):
    temp = MyDescriptor(37)  # Descriptor is attached at class definition time


body = Model()
```

```
__init__ of MyDescriptor called with initial value:  37
Setting self.val to  37
```

When we created an instance of *Model* class, the *MyDescriptor* was initiated. When *MyDescriptor* was initiated, its `__init__` method was called and got the string printed.

```
print(body.temp)  # a function call from `MyDescriptor` class is hiding here
```

```
<__main__.Model object at 0x7f136954a7c0> <class '__main__.Model'>
Getting  self.val:  37
37
```

Above when we ran *body.temp*, the `__get__` method of *MyDescriptor* class for *temp* attribute ran.

```
body.temp = 38  # a function call is hiding here
```

```
Setting self.val to  38
```

```
print(body.temp)
```

```
<__main__.Model object at 0x7f136954a7c0> <class '__main__.Model'>
Getting  self.val:  38
38
```

Thus we see when we get the value of attribute `temp`, the method `__get__` in *MyDescriptor* gets executed and similarly when we set a value to attribute *temp*, the method `__set__` in *MyDescriptor* gets executed.

In `__get__` method of *MyDescriptor* class, instance is *body* and and owner is *Model*.

If we want to know what attributes are stored in `__dict__` of class and instance, we can do as following

```
print(body.__dict__)
```

```
{}
```

The `__get__` and `__set__` of descriptor can be applied only on those attributes which are present in `__dict__` of owner class i.e. *Model* in this case.

```
# alternative to print(Model.__dict__)


for key, val in Model.__dict__.items():
    print(key, ': ', val)
```

```
__module__ :  __main__
temp :  <__main__.MyDescriptor object at 0x7f136957de80>
__dict__ :  <attribute '__dict__' of 'Model' objects>
__weakref__ :  <attribute '__weakref__' of 'Model' objects>
__doc__ :  None
```

alternative to print(MyDescriptor.__dict__)

```
for key, val in MyDescriptor.__dict__.items():
    print(key, ': ', val)
```

```
__module__ :  __main__
__doc__ :
    Basic descriptor to set and get value.

__init__ :  <function MyDescriptor.__init__ at 0x7f1369535ee0>
__get__ :  <function MyDescriptor.__get__ at 0x7f1369535790>
__set__ :  <function MyDescriptor.__set__ at 0x7f1369535dc0>
__dict__ :  <attribute '__dict__' of 'MyDescriptor' objects>
__weakref__ :  <attribute '__weakref__' of 'MyDescriptor' objects>
```

We can call *get* from class and its instance but *set* can and should only be called from instance. If we do it from class, this means overriding descriptor.

```
print(Model.temp)
```

```
None <class '__main__.Model'>
Getting  self.val:  38
38
```

```
Model.temp = "useless"
print(Model.temp)
```

```
useless
```

```
print(body.temp)
```

```
useless
```

This means we should do some type checking before assigning a value to an attribute. Consider *descriptor* from another angle below.

```
class LazyDescriptor(object):
    def __init__(self, name, inival):
        self._val = inival
        self.name = name

    def __get__(self, instance, owner):
        print('get in descriptor called')
        instance.__dict__[self.name] = self._val
        return self._val
```

(continues on next page)

(continued from previous page)

```
class Model(object):
    temp = LazyDescriptor("temp", 37)


body = Model()
```

```
print(body.temp)
```

```
get in descriptor called
37
```

Above we we ran *body.temp*, the `__get__` method of descriptor was called.

```
print(body.temp)
```

```
37
```

So the first time we referenced temp, it called the descriptor but not the second time. Let's look at the *__dict__* for better understanding.

```
body = Model()
print(body.__dict__)
```

```
{}
```

```
print(body.temp)
```

```
get in descriptor called
37
```

```
print(body.__dict__)
```

```
{'temp': 37}
```

```
print(body.temp)
```

```
37
```

```
print(body.__dict__)
```

```
{'temp': 37}
```

So when we tried to access the value of *x* for the first time, the *key* was not in `object.__dict__` so the descriptor's `__get__` was called but when it is already present, the `__get__` from descriptor was not called. This is because of order in which python looks for attributes of objects. For complete sequence of rules see this link. We can achieve exactly same by another way as well.

```
class LazyProperty(object):
    def __init__(self, val):
        self._val = val
        self.name = val.__name__

    def __get__(self, instance, owner):
        print("get in descriptor called")
        result = self._val(instance)
        instance.__dict__[self.name] = result
        return result


class Model(object):
    @LazyProperty
    def temp(self):
        return 42


body = Model()
```

```
print(body.temp)
```

```
get in descriptor called
42
```

```
print(body.temp)
```

```
42
```

Usage cases

Suppose we define a class which takes the *name*, *weight* and *height* as input/for initiation and has a method to calculate body mass index i.e. *bmi*.

```
class Insan:
    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight  # in kg
        self.height = height  # in meters

    def bmi(self):
        return self.weight / self.height ** 2


ali = Insan('ali', 78, 1.7)
ali.bmi()
```

```
26.989619377162633
```

The problem with the above code is that one can assign negative values to weight.

```
ali.weight = -10
ali.bmi()
```

```
-3.4602076124567476
```

Definitely it is wrong and we should perform some checks before setting the new value. We can do it by using *property*

```
class Insan:
    def __init__(self, name, weight, height):
        self.name = name
        self._weight = weight  # in kg
        self.height = height  # in meters

    @property
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, value):
        if value < 0:
            raise ValueError('weight cannot be negative.')
        self._weight = value

    def bmi(self):
        return self.weight / self.height ** 2


ali = Insan('ali', 78, 1.7)

# uncomment following line
# ali.weight = -80  # ValueError
ali.bmi()
```

```
26.989619377162633
```

Thus upon negative weight, it threw error. But *height* can still be assigned a negative value.

```
ali = Insan('ali', 78, 1.7)
ali.height = -1.8
print(ali.height)
```

```
-1.8
```

Let's make use of *property* once more.

```
class Insan:
    def __init__(self, name, weight, height):
        self.name = name
        self._weight = weight  # in kg
        self._height = height  # in meters

    @property
```

(continues on next page)

(continued from previous page)

```
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, value):
        if value < 0:
            raise ValueError('weight cannot be negative.')
        self._weight = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value < 0:
            raise ValueError('height cannot be negative.')
        self._height = value


ali = Insan('Ali', 78, 1.7)

# uncomment following line
# ali.weight = -80  # ValueError
```

But we are repeating our code. Both the properties are essentially doing same thing i.e. throwing errors on negative value assignment, so remember our code should be DRY (do not repeat yourself) To helps us, python has the concept of *descriptors*. We can define a descriptor which can have `set`, `get` and `del` methods. The following code defines the descriptor *NonNegative*. Then inside class *Insan*, we define class attributes and bind them with the descriptor thus making sure that these attributes will always be non-negative otherwise an error will be thrown.

```
class NonNegative:
    def __init__(self, name):
        # the name attribute is needed because when the NonNegative object is
        # created , the assignment to attribute named weight/height hasn't
        # happen yet. Thus, we need to explicitly pass the name weight/height to the
        # initializer of the object to use as the key for the instance's __dict__.
        self.name = name

    def __get__(self, instance, owner):
        # we need to reach into the __dict__ object directly, because the
        # builtins would be intercepted
        #  by the descriptor protocols too and cause the RecursionError.
        return instance.__dict__[self.name]  # getattr(instance, self._name)

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError("{} Cannot be negative.".format(self.name))
        # instead of using builtin function getattr and setattr, we need to reach
        # into the __dict__ object directly, because the builtins would be intercepted
        #  by the descriptor protocols too and cause the RecursionError.
        instance.__dict__[self.name] = value  # setattr(instance, self._name, value)
```

(continues on next page)

(continued from previous page)

```
class Insan:
    weight = NonNegative('weight')
    height = NonNegative('height')

    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight  # in kg
        self.height = height  # in meters

    def bmi(self):
        return self.weight / self.height ** 2


ali = Insan('Ali', 78, 1.7)
ali.bmi()
```

```
26.989619377162633
```

Now we can not assign negative values to attributes `weight` and `height` of class `Insan`.

```
# uncomment following line
# ali.weight = -80 # ValueError: Cannot be negative
```

```
# uncomment following line
# ali.height = -1.8  # ValueError: Cannot be negative
```

```
### In python 3.6+

# The `descriptor` definition in python 3.6+ is more flexible.
```

```
class NonNegative:
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('{} Cannot be negative.'.format(self.name))
        instance.__dict__[self.name] = value

    # __set_name__ is called at the time the owning class owner is created.
    # The descriptor has been assigned to name. With this protocol, we can now
    # remove the __init__  and bind the attribute name to the descriptor
    def __set_name__(self, owner, name):
        self.name = name


class Insan:
    weight = NonNegative()
    height = NonNegative()
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name, weight, height):
        self.name = name
        self.weight = weight  # in kg
        self.height = height  # in meters

    def bmi(self):
        return self.weight / self.height ** 2


ali = Insan('Ali', 78, 1.7)
ali.bmi()
```

```
26.989619377162633
```

```
# uncomment following line
# ali.weight = -80  # ValueError: Cannot be negative
```

```
# uncomment following line
# ali.height = -1.8  # ValueError: Cannot be negative
```

Let's say, we want to calculate a new quantity say *bmi* which is multiplication of *BMI* with *temperature* in Celsius. We can define a property to convert the temperature into Celsius, in case the temperature is provided in Fahrenheit.

```
class NonNegative:

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('{} Cannot be negative.'.format(self.name))
        instance.__dict__[self.name] = value

    def __set_name__(self, owner, name):
        self.name = name


class Insan:
    weight = NonNegative()
    height = NonNegative()

    def __init__(self, name, weight, height, temp_f):
        self.name = name
        self.weight = weight  # in kg
        self.height = height  # in meters
        self.fahrenheit = temp_f

    @property
    def celsius(self):
        return 5 * (self.fahrenheit - 32) / 9.0
```

(continues on next page)

(continued from previous page)

```
    @celsius.setter
    def celsius(self, val):
        self.fahrenheit = 32 + 9 * val / 5.0

    def bmit(self):
        return self.weight / self.height ** 2 * self.celsius


ali = Insan('Ali', 78, 1.7, 98.2)
ali.bmit()
```

```
992.6182237600924
```

```
print(ali.celsius)
```

```
36.77777777777778
```

But we can also define it as *descriptor* as follows. Furthermore we are also performing non-negative check in this descriptor as well.

```
class Celsius:

    def __get__(self, instance, owner):
        return 5 * (instance.fahrenheit - 32) / 9

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('{} Cannot be negative.'.format(self.name))
        instance.fahrenheit = 32 + 9 * value / 5

    def __set_name__(self, owner, name):
        self.name = name


class NonNegative:

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('{} Cannot be negative.'.format(self.name))
        instance.__dict__[self.name] = value

    def __set_name__(self, owner, name):
        self.name = name


class Insan:
    weight = NonNegative()
```

(continues on next page)

(continued from previous page)

```
    height = NonNegative()

    celsius = Celsius()

    def __init__(self, name, weight, height, temp_f):
        self.name = name
        self.weight = weight  # in kg
        self.height = height  # in meters
        self.fahrenheit = temp_f  # temperature in fahrenheit

    def bmit(self):
        return self.weight / self.height ** 2 * self.celsius


ali = Insan('Ali', 78, 1.7, 98.2)
ali.bmit()
```

```
992.6182237600924
```

```
print(ali.fahrenheit)
```

```
98.2
```

```
print(ali.celsius)
```

```
36.77777777777778
```

```
# uncomment following line
# ali.celsius = -30  # ValueError
```

Caveat

Because the descriptors are linked with class and not with instance, so when we create a new instance, the values get overridden by new instance if they are not linked with instance.

```
class Descriptor:
    def __init__(self):
        self.__temp = 0

    def __get__(self, instance, owner):
        return self.__temp

    def __set__(self, instance, value):
        if isinstance(float(value), float):
            print(value)
        else:
            raise TypeError("Body Temperature must be float or integer")

        if value < 20:
```

(continues on next page)

(continued from previous page)

```
            raise ValueError("Body Temperature can never be less than 20")

        self.__temp = value

    def __set_name__(self, owner, name):
        self.name = name


class Model:
    temp = Descriptor()

    def __init__(self, name, weight, temp):
        self._name = name
        self.weight = weight
        self.temp = temp

    def __str__(self):
        return "{0} with weight {1} has body temperature {2} Celsius.".format(self._name,
↪ self.weight, self.temp)


body1 = Model("Ali", 80, 40)
print(body1)
```

```
40
Ali with weight 80 has body temperature 40 Celsius.
```

```
print(body1.__dict__)
```

```
{'_name': 'Ali', 'weight': 80}
```

```
body2 = Model("Hasan", 75, 37)
print(body2)
```

```
37
Hasan with weight 75 has body temperature 37 Celsius.
```

```
print(body1)
```

```
Ali with weight 80 has body temperature 37 Celsius.
```

The solution is to bind the attribute with instance in descriptor as shown below.

```
class Descriptor:
    def __init__(self):
        self.__temp = 0

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]
```

(continues on next page)

(continued from previous page)

```
    def __set__(self, instance, value):
        if isinstance(float(value), float):
            print(value)
        else:
            raise TypeError("Body Temperature must be float or integer")

        if value < 20:
            raise ValueError("Body Temperature can never be less than 20")

        instance.__dict__[self.name] = value

    def __set_name__(self, owner, name):
        self.name = name


class Model:
    temp = Descriptor()

    def __init__(self, name, weight, temp):
        self.name = name
        self.weight = weight
        self.temp = temp

    def __str__(self):
        return "{0} with weight {1} has body temperature {2} Celsius.".format(self.name,␣
↪self.weight, self.temp)


body1 = Model("Ali", 80, 40)
print(body1)
```

```
40
Ali with weight 80 has body temperature 40 Celsius.
```

```
print(body1.__dict__)
```

```
{'name': 'Ali', 'weight': 80, 'temp': 40}
```

```
body2 = Model("Hasan", 75, 37)
print(body2)
```

```
37
Hasan with weight 75 has body temperature 37 Celsius.
```

```
print(body1)
```

```
Ali with weight 80 has body temperature 40 Celsius.
```

Using WeakKeyDictionary

Usually the attributes from descriptors are saved in `WeakKeyDictionary`. The above code can be implemented using `WeakKeyDictionary` as shown below

```
from weakref import WeakKeyDictionary


class Descriptor:
    def __init__(self):
        self.data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        return self.data[instance]

    def __set__(self, instance, value):
        if isinstance(float(value), float):
            print(value)
        else:
            raise TypeError("Body Temperature must be float or integer")

        if value < 20:
            raise ValueError("Body Temperature can never be less than 20")

        self.data[instance] = value

    def __set_name__(self, owner, name):
        self.name = name


class Model:
    temp = Descriptor()

    def __init__(self, name, weight, temp):
        self.name = name
        self.weight = weight
        self.temp = temp

    def __str__(self):
        return "{0} with weight {1} has body temperature {2} Celsius.".format(self.name,␣
↪self.weight, self.temp)


body1 = Model("Ali", 80, 40)
print(body1)
```

```
40
Ali with weight 80 has body temperature 40 Celsius.
```

```
body2 = Model("Hasan", 75, 37)
print(body2)
```

```
37
Hasan with weight 75 has body temperature 37 Celsius.
```

```
print(body1)
```

```
Ali with weight 80 has body temperature 40 Celsius.
```

References:

The material in this lesson is inspired from following posts

- talk on descriptors
- Python course eu website
- Encapsulation with descriptors
- Some great answers on stackoverflow
- A post by Daw Ran Liou
- DataCamp

Total running time of the script: (0 minutes 0.021 seconds)

3.13 inheritance

Inheritance means one class can inherit `attributes` and `methods` of parent class.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def say_salam(self):
        print('Salam, my name is', self.name)


class Muslim(Human):
    pass


ali = Muslim('Ali')
ali.say_salam()
```

```
Salam, my name is Ali
```

The `Muslim` class itself does not have a method `say_salam` but since it is inheriting from `Human` class and `say_slam` exists in `Human` class, we do end up calling `say_salam` method fo `Human` class.

We can check the type of the `ali` which is an object and is an `instance` of class `Muslim`. There are two ways to verify this.

```
isinstance(ali, Muslim), type(ali) == Muslim
```

```
(True, True)
```

However, `isinstance` checks the parent classes as well but `type` function does not do so.

```
isinstance(ali, Human), type(ali) == Human
```

```
(True, False)
```

```
arjun = Human('arjun')

isinstance(arjun, Human), isinstance(arjun, Muslim)
```

```
(True, False)
```

So it is always safe to check the type of an object by `instance`.

Inheritance can occur between more than two classes as well. In following, `Pakistani` class inherits from *Muslim* class which itself inherits from `Human` class.

```
class Human(object):

    def __init__(self, name):
        self.name = name


class Muslim(Human):
    pass


class Pakistani(Muslim):
    pass


ali = Pakistani('ali')

isinstance(ali, Human)
```

```
True
```

We can override the methods of parent or super class in its child class by simply redefining the `method`.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):

    def introduction(self):
```

(continues on next page)

(continued from previous page)

```
        print('Salam, my name is', self.name)


ali = Muslim("ali")

ali.introduction()
```

```
Salam, my name is ali
```

```
Human.introduction(ali)
```

```
Hello, my name is ali
```

If in a child class we want to call/use method from a parent/super class, we can call method from parent/super class using the *super()* function as following.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):

    def introduction(self):
        print("I will call introduction method of parent class")
        super().introduction()


ali = Muslim("ali")

ali.introduction()
```

```
I will call introduction method of parent class
Hello, my name is ali
```

Above, the `introduction` method of `Muslim` class calls the `introduction` method of `Human`.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):
```

(continues on next page)

(continued from previous page)

```
    def introduction(self):
        print("salam: my name is ", self.name)


class Pakistani(Muslim):

    def introduction(self):
        print("I will call introduction method of parent class")
        super().introduction()


ali = Pakistani("ali")

ali.introduction()
```

```
I will call introduction method of parent class
salam: my name is  ali
```

If a method is not present in super / parent class , python will keep on looking in all parent / super classes in the hierarchy until it finds the method or attribute being called.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):
    pass


class Pakistani(Muslim):

    def introduction(self):
        print("I will call introduction method of parent class")
        super().introduction()


ali = Pakistani("ali")

ali.introduction()
```

```
I will call introduction method of parent class
Hello, my name is ali
```

```
class Human(object):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):
    pass


class Pakistani(Muslim):
    pass


ali = Pakistani("ali")

ali.introduction()
```

```
Hello, my name is ali
```

However, if the called method does not exist in any of the parent classes, then python will throw `AttributeError`.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction1(self):
        print('Hello, my name is', self.name)


class Muslim(Human):
    pass


class Pakistani(Muslim):
    pass


ali = Pakistani("ali")

# uncomment following line
# ali.introduction()  # AttributeError
```

If in a child class , we want to implement a method of a parent class but not of immediate parent class, we can directly call that method.

Below, `Pakistani` class is inheriting from `Muslim` but in introduction method of `Pakistani`, we don't want to call `introduction` method of `Muslim` class rather we want to call `introduction` method of `Human` class.We can do this by `Human.introduction()`

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):

    def introduction(self):
        print('Salam, my name is', self.name)


class Pakistani(Muslim):

    def introduction(self):
        Human.introduction(self)
        print("I am a Pakistani. ")


ali = Pakistani("ali")

ali.introduction()
```

```
Hello, my name is ali
I am a Pakistani.
```

One of the function of method overriding is to implement abstract methods.

```
class Human(object):

    def nationality(self):
        raise NotImplementedError("implement this method in child class")


class Pakistani(Human):

    def nationality(self):
        return "pakistani"


class Iranian(Human):
    def nationality(self):
        return "iranian"


class Israili(Human):
    pass


ali = Pakistani()
```

(continues on next page)

(continued from previous page)

```
ali.nationality()
```

```
'pakistani'
```

```
armaghan = Iranian()
armaghan.nationality()
```

```
'iranian'
```

```
yakov = Israili()

# uncomment following line
# yakov.nationality()  # NotImplementedError
```

We can enlist all base classes in a hierarchy of given class.

```
class Human(object):
    pass


class Muslim(Human):
    pass


class Pakistani(Human):
    pass


class Punjabi(Pakistani):
    pass


print(Punjabi.__mro__)
```

```
(<class '__main__.Punjabi'>, <class '__main__.Pakistani'>, <class '__main__.Human'>,
↪<class 'object'>)
```

```
print(Muslim.__mro__)
```

```
(<class '__main__.Muslim'>, <class '__main__.Human'>, <class 'object'>)
```

We can initiate a child class with additional arguments than required to initiate the parent class. For this we have to overwrite `__init__()` method of the child class.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
```

(continues on next page)

(continued from previous page)

```
        print('Hello, my name is', self.name)


class Muslim(Human):

    def __init__(self, name, sect):
        self.sect = sect
        super(Muslim, self).__init__(name)

    def introduction(self):
        print('Salam, my name is {} and my sect is {}'.format(self.name, self.sect))


# Uncomment following line
# ali = Muslim("ali")  # TypeError
```

Above, the `Muslim` class now requires two arguments for initiation i.e. *name* ` and *sect*. We did not provide value for `sect`.

```
ali = Muslim("ali", None)
ali.introduction()
```

```
Salam, my name is ali and my sect is None
```

Instead of using `super(ChildClass, self).__init__()` we can also simple say `super().__init__()`.

```
class Human(object):

    def __init__(self, name):
        self.name = name

    def introduction(self):
        print('Hello, my name is', self.name)


class Muslim(Human):

    def __init__(self, name, sect):
        self.sect = sect
        super().__init__(name)

    def introduction(self):
        print('Salam, my name is {} and my sect is {}'.format(self.name, self.sect))


ali = Muslim("ali", None)
ali.introduction()
```

```
Salam, my name is ali and my sect is None
```

```
type(ali.sect)
```

```
kapil = Human('kapil')

# uncomment following line
# kapil.sect  # AttributeError
```

Since `kapil` is an instance of `Human` class which does not have any attribute named `sect`, thus we could not access this attribute.

multiple-inheritance

A class can also inherit from multiple classes and this is called `multiple-inheritance`.

```
class Spirit(object):
    chastity = 'NaN'
    bravery = 'NaN'
    tolerance = 'NaN'


class Body(object):
    weight = 20


class Human(Spirit, Body):
    pass


ali = Human()
print(ali.chastity)
```

```
NaN
```

```
print(ali.weight)
```

```
20
```

If an attribute of the child class is called, it is first looked in the child class, then in the first parent class and then in second parent class.

```
class Spirit(object):
    chastity = 'NaN'
    bravery = 'NaN'
    identity = 'NaN'


class Body(object):
    identity = 'name'


class Human(Spirit, Body):
    pass
```

(continues on next page)

(continued from previous page)

```
ali = Human()
print(ali.identity)
```

```
NaN
```

Above, `identity` attribute exists for `Body` and `Spirit` class but since `Spirit` class comes first, so `NaN` is printed which is value of `identity` attribute of `Spirit` class. Sometimes, we may want to initialize both parent classes with separate initializing arguments. In following, `Body` class requires two initiating arguments while `Spirit` class can be initialized with as many arguments as possible.

```
class Spirit(object):
    def __init__(self, **kwargs):
        print("initializing spirit")
        for k, v in kwargs.items():
            print('setting: ', k, 'to ', v)
            setattr(self, k, v)
        print("finished initializing spirit")


class Body(object):
    def __init__(self, weight, height):
        print("initializing body")
        self.weight = weight
        self.height = height
        print("finished initializing body")


class Human(Spirit, Body):
    def __init__(self, name, weight, height, **kwargs):
        print("initializing Human")
        self.name = name
        Spirit.__init__(self, **kwargs)
        Body.__init__(self, weight, height)


ali = Human('ali', 60, 175, chastity='NotANumber')

print("Name: ", ali.name)
print("Weight: ", ali.weight)
print("Height: ", ali.height)
print("Chastity: ", ali.chastity)
```

```
initializing Human
initializing spirit
setting:  chastity to  NotANumber
finished initializing spirit
initializing body
finished initializing body
Name:  ali
Weight:  60
Height:  175
```

(continues on next page)

(continued from previous page)

```
Chastity:  NotANumber
```

Instead of initializing both parent classes separately, call to `super()` method will suffice to initiate all parent classes at once.

```
class Spirit(object):
    def __init__(self, **kwargs):
        print("initializing spirit")
        for k, v in kwargs.items():
            print('setting: ', k, 'to ', v)
            setattr(self, k, v)
        print("finished initializing spirit")


class Body(object):
    def __init__(self, weight, height):
        print("initializing body")
        self.weight = weight
        self.height = height
        print("finished initializing body")


class Human(Spirit, Body):
    def __init__(self, name, weight, height, **kwargs):
        print("initializing Human")
        self.name = name
        super().__init__(weight=weight, height=height, **kwargs)


ali = Human('ali', 60, 175, chastity='NotANumber', honesty="undefined")

print("Name: ", ali.name)
print("Weight: ", ali.weight)
print("Height: ", ali.height)
print("Chastity: ", ali.chastity)
```

```
initializing Human
initializing spirit
setting:  weight to  60
setting:  height to  175
setting:  chastity to  NotANumber
setting:  honesty to  undefined
finished initializing spirit
Name:  ali
Weight:  60
Height:  175
Chastity:  NotANumber
```

Total running time of the script: (0 minutes 0.014 seconds)

3.14 __call__

This lesson shows the usage of `__call__` method of a class.

Suppose we have a function named *human*.

```
def human(age):
    print(f"my age is {age}")
    return
```

Since *human* is a function, we can **call** this function using `()`.

```
human(12)
```

```
my age is 12
```

What if we want a class (or an instance of a class) to behave like function. To be specific if we want to use an instance of a class as a function, so that when we can call this instance, we will write `__call__` method of the class.

```
class Human:

    def __init__(self, age):
        self.age = age

    def __call__(self):
        print(f"my age is {self.age}")
        return
```

The *Human* class has two (user defined) methods i.e. `__init__` and `__call__`. The `__init__` method is used/called when we initialize th class and create its instance.

```
human = Human(12)
```

Now if we use the instance *human* as function i.e. if we call *human* using `()`, the `__call__` method will be executed.

```
human()
```

```
my age is 12
```

The `__call__` can be defined to take any keyword, non-keyword, optional or obligatory arguments.

```
class Human:

    def __init__(self, age):
        self.age = age

    def __call__(self, characteristic):
        print(f"I am {characteristic}")
        return


human = Human(63)
human("mortal")
```

```
I am mortal
```

```
class Human:

    def __init__(self, age):
        self.age = age

    def __call__(self, *args, **kwargs):
        print(f"{args} {kwargs}")
        return


human = Human(63)
human()
```

```
() {}
```

```
human(1)
```

```
(1,) {}
```

```
human(1, a=2)
```

```
(1,) {'a': 2}
```

Without writing, `__call__` method for the *Human* class, we wouldn't be able to call the *Human* class or its instance *human*.

Question: In the code `Human(1)()`, what does the second braces `()`, those which come after **(1)** represent?

```
class Human:

    def __init__(self, age):
        self.age = age


human = Human(2)

# uncomment the following line
# human()  # -> TypeError: 'Human' object is not callable.
```

Total running time of the script: (0 minutes 0.003 seconds)

3.15 __getattr__

This lesson describes the usage of `__getattr__`

If a `class` (more correctly ab object) does not have an attribute and we try to access this attribute we will get `AttributeError`.

```
class Human:
    pass


h = Human()

# uncomment following 1 line
# h.horns
```

The *Human* class does not have an attribute *horns* and therefore when we run *h.horns*, we will get AttributeError

However, if we want to avoid such an error, we can overwrite `__getattr__` method of the class. This method must take one input argument.

```
class Human:
    def __getattr__(self, item):
        print(f"attribute {item} has not been set to Human")
        return


h = Human()
print(h.horns)
```

```
attribute horns has not been set to Human
None
```

When python tries to search attributes of a class, then `__getattr__` method is called at the end of its search. If this method is not overwritten by the user, python will raise `AttributeError`, as it was done earlier.

One advantage/usage of this method is what we can call *dynamic attribute creation*.

```
TempUnitConverter = {
    "FAHRENHEIT": {
        "Fahrenheit": lambda fahrenheit: fahrenheit,  # fahrenheit to Centigrade
        "Kelvin": lambda fahrenheit: [(x + 459.67) * 5/9 for x in fahrenheit],  #␣
↪fahrenheit to kelvin
        "Centigrade": lambda fahrenheit: [(x - 32.0) / 1.8 for x in fahrenheit]  #␣
↪fahrenheit to Centigrade
    },
    "KELVIN": {
        "Fahrenheit": lambda kelvin: [x * 9/5 - 459.67 for x in kelvin],  # kelvin to␣
↪fahrenheit
        "Kelvin": lambda k: k,      # Kelvin to Kelvin
        "Centigrade": lambda kelvin: [x - 273.15 for x in kelvin]  # kelvin to␣
↪Centigrade}
    },
    "CENTIGRADE": {
        "Fahrenheit": lambda centigrade: [x * 1.8 + 32.0 for x in centigrade],  #␣
↪Centigrade to fahrenheit
```

(continues on next page)

(continued from previous page)

```
        "Kelvin": lambda centigrade: [x + 273.15 for x in centigrade],  # Centigrade to␣
↪kelvin
        "Centigrade": lambda centigrade: centigrade
    }
}


class Temperature(object):
    """
    The idea is to write the conversion functions in a dictionary and
    then dynamically create attribute if the attribute
    is present in converter as key otherwise raise WongUnitError.
    converts temperature among units [kelvin, centigrade, fahrenheit]

    """

    def __init__(self, val, input_unit):
        self.val = val

        self.input_unit = input_unit

    def __getattr__(self, out_unit):
        # pycharm calls this method for its own working, executing default behaviour at␣
↪such calls
        if out_unit.startswith('_'):
            return self.__getattribute__(out_unit)
        else:
            if out_unit not in TempUnitConverter[self.input_unit]:
                raise ValueError(f"output in {out_unit} is not allowed. Allowed units␣
↪are: ", self.allowed)

            val = TempUnitConverter[self.input_unit][str(out_unit)](self.val)
            return val

    @property
    def allowed(self):
        return list(list(TempUnitConverter.values())[0].keys())

    @property
    def input_unit(self):
        return self._input_unit

    @input_unit.setter
    def input_unit(self, in_unit):
        if in_unit.upper() == 'CELSIUS':
            in_unit = 'CENTIGRADE'
        if in_unit.upper() not in TempUnitConverter:
            raise ValueError(f"Input in {in_unit} is not allowed", self.allowed)
        self._input_unit = in_unit.upper()


temp = [i for i in range(10)]
```

(continues on next page)

(continued from previous page)

```
T = Temperature(temp, 'Centigrade')

print(T.Kelvin)
```

```
[273.15, 274.15, 275.15, 276.15, 277.15, 278.15, 279.15, 280.15, 281.15, 282.15]
```

```
print(T.Fahrenheit)
```

```
[32.0, 33.8, 35.6, 37.4, 39.2, 41.0, 42.8, 44.6, 46.4, 48.2]
```

```
print(T.Centigrade)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Above we did not explicitly defined `Kelvin`, `Fahrenheit` or `Centigrade` attributes of the `Temperature` class. but these attributes are created after `__getattr__` method is called.

```
T = Temperature(temp, 'Fahrenheit')
print(T.Centigrade)
print(T.Kelvin)
```

```
[-17.77777777777778, -17.22222222222222, -16.666666666666668, -16.11111111111111, -15.
↪555555555555555, -15.0, -14.444444444444445, -13.88888888888889, -13.333333333333332, -
↪12.777777777777777]
[255.3722222222222, 255.92777777777778, 256.48333333333335, 257.0388888888889, 257.
↪59444444444443, 258.15, 258.7055555555555, 259.2611111111111, 259.81666666666666, 260.
↪3722222222222]
```

```
T = Temperature(temp, 'Kelvin')
print(T.Centigrade)
print(T.Fahrenheit)
```

```
[-273.15, -272.15, -271.15, -270.15, -269.15, -268.15, -267.15, -266.15, -265.15, -264.
↪15]
[-459.67, -457.87, -456.07, -454.27000000000004, -452.47, -450.67, -448.87, -447.07, -
↪445.27000000000004, -443.47]
```

Questions:

- Why *Temperature(temp, 'Celsius').Kelvin* works but not *Temperature(temp, 'Celsius').Celsius*?
- Change the *Temperature* class so that T.centigrade gives same answer as that of *T.Centigrade*.
- Change the *Temperature* class so that *T.Celsius* also works.

Total running time of the script: (0 minutes 0.004 seconds)

3.16 magic methods

This file describes the so called magic methods in python. Magic methods are those methods which start with double underscore __ sign. We have already seen some of the magic methods such as `__init__` in *3.5 init method* , `__call__` in *3.14 __call__* and about `__repr__` and `__str__` in *3.6 str and repr methods* lessons. Here we will cover some more.

`__add__`

This method determines the behavior when addition is performed on the instance of its class. Thus, using `__add__` method of a class, we can define how the addition on the instance of this class will work. For example, in class *NonSenseInteger* below, we are defining `__add__` method, so any instance of *NonSenseInteger* class will behave the way we are defining in `__add__` method.

```
import os


class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __add__(self, other):
        return self.value - other


ns_int = NonSenseInteger(10)

print(ns_int + 5)
```

```
5
```

Above we have defined that addition will work as subtraction.

```
print(5 + ns_int)
```

```
15
```

However, above, when the instance of class is on right side of addition operation, the addition did not happened the way we defined in `__add__` method.

`__radd__`

The `__add__` method does not determines the addition behavior of a class when the instance of the class is on right side of + operator. In order to overwrite this behavior i.e., the working of addition operation when the instance of class is on right side of +, we have to write `__radd__` method.

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

```

(continues on next page)

(continued from previous page)

```
    def __add__(self, other):
        return self.value - other

    def __radd__(self, other):
        return self.value * other


ns_int = NonSenseInteger(10)

print(ns_int + 5)
print(5 + ns_int)
```

```
5
50
```

Above we see that when *ns_int* was on left side, subtraction was performed as we defined in `__add__` method and when *ns_int* was on right side, multiplication was performed as we defined inside `__radd__` method.

`__mul__`

This method determines the behavior when multiplication is performed on the instance of its class.

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __mul__(self, other):
        return self.value + other


ns_int = NonSenseInteger(10)

print(ns_int * 5)
```

```
15
```

Although 10 * 5 is 50, but we got 15, because we modified the multiplication behavior of our *NoneSenseInteger* class.

```
print(5 * ns_int)
```

```
50
```

__rmul__

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __mul__(self, other):
        return self.value + other

    def __rmul__(self, other):
        return self.value - other


ns_int = NonSenseInteger(10)
print(ns_int * 5)
print(5 * ns_int)
```

```
15
5
```

__sub__

This method determines the behavior when subtraction operation is performed on the instance of its class.

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __sub__(self, other):
        return self.value + other


ns_int = NonSenseInteger(10)
print(ns_int - 5)
```

```
15
```

```
print(5 - ns_int)
```

```
-5
```

__rsub__

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __sub__(self, other):
        return self.value + other

    def __rsub__(self, other):
        return self.value * other


ns_int = NonSenseInteger(10)
print(ns_int - 5)
print(5 - ns_int)
```

```
15
50
```

__truediv__

This method determines the behavior when division operation is performed on the instance of the class.

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __truediv__(self, other):
        return self.value * other


ns_int = NonSenseInteger(10)
print(ns_int / 5)
```

```
50
```

```
print(5 / ns_int)
```

```
0.5
```

__rtruediv__

```
class NonSenseInteger(int):

    def __init__(self, value):
         self.value = value

    def __truediv__(self, other):
        return self.value * other

    def __rtruediv__(self, other):
        return self.value + other


ns_int = NonSenseInteger(10)

print(ns_int / 5)
print(5 / ns_int)
```

```
50
15
```

__enter__ and __exit__

These methods are used by the context manager i.e. `with`. They are executed/called when we 'enter' and 'exit' the context manager.

```
class Insan:
    def __init__(self, name, year, age):
        self.name = name
        self.year = year
        self.age = age

    def __enter__(self):
        print(f"{self.name} was born in year {self.year}")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"{self.name} lived until {self.year + self.age} year")
        return

    def married(self, spouse_name:str):
        print(f"{self.name} married with {spouse_name}")
        return
```

```
with Insan('Ali', 600, 63) as person:
    print("entered")
```

```
Ali was born in year 600
entered
Ali lived until 663 year
```

If you note the print order of strings, you will find out that `__enter__` method was executed before `print()` function was called. Similarly, `__exit__` method was executed after `print()` function was called i.e. at the time of exiting the context manager.

what if we implment only `__exit__` and not `__enter__`?

```
class Insan:
    def __init__(self, name, year, age):
        self.name = name
        self.year = year
        self.age = age

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"{self.name} lived until {self.year + self.age} year")
        return

    def married(self, spouse_name:str):
        print(f"{self.name} married with {spouse_name}")
        return
```

```
ali = Insan('Ali', 600, 63)
print(type(ali))
```

```
<class '__main__.Insan'>
```

```
# uncomment following three lines
# with Insan('Ali', 600, 63) as person:
#     print("entered")
#     person.married('Falima') # -> AttributeError: __enter__
```

The error message shows that it is not possible to use *Insan* class with context manager without implementing `__enter__` method for this class. This is because when we say *with Insan('Ali', 600, 63) as person:*, the `__enter__` method of *Insan* class is called implicitly. When this method does not exist, we get the error as shown above.

Same is true if we implement only `__enter__` method and not `__exit__` method.

```
class Insan:
    def __init__(self, name, year, age):
        self.name = name
        self.year = year
        self.age = age

    def __enter__(self):
        print(f"{self.name} was born in year {self.year}")
        return self

    def married(self, spouse_name:str):
        print(f"{self.name} married with {spouse_name}")
        return
```

```
# uncomment following three lines
# with Insan('Ali', 600, 63) as person:
#     print("entered")
#     person.married('Falima') # -> AttributeError: __enter__
```

Other than that, we can still use this class as normal python class.

```
ali = Insan('Ali', 600, 63)
print(ali.age)
```

```
63
```

__iter__ and __next__

The __next__ magic method determines what will happen when we call next function on the instance of the class.

```
class Insan:
    def __init__(self, num_child):
        self.children = [f"child_{i}" for i in range(num_child)]
        self.index = 0

    def __next__(self):
        item = self.children[self.index]
        self.index += 1
        return item


ali = Insan(2)

print(ali.children)
```

```
['child_0', 'child_1']
```

```
next(ali)
```

```
'child_0'
```

```
next(ali)
```

```
'child_1'
```

If we call the next function again on *ali*, we will get IndexError due to what is happening inside __next__ method above.

```
# uncomment following line
# next(ali)
```

Although, we can apply next function on *ali* but we can still not use it in a for loop.

```
ali = Insan(2)

# uncomment following two lines
# for child in ali:
#     print
```

This is because *ali* is not an iterable and we can verify it as below

```
import collections

isinstance(ali, collections.abc.Iterable)
```

```
False
```

Since *ali* is not an "iterable", therefore we can not use it in `for` loop. The reason is that the `for` loop requests an iterator from the iterable object, and then calls `__next__` on that iterable until it hits the `StopIteration` exception. This happens under the surface which is also the reason why we would want iterators to implement the `__iter__` as well.

Question: why the code `isinstance(ali, collections.abc.Iterator)` returns False?

```
class Insan:
    def __init__(self, num_child):
        self.children = [f"child_{i}" for i in range(num_child)]
        self.index = 0

    def __next__(self):
        item = self.children[self.index]
        self.index += 1
        return item

    def __iter__(self):
        return self


ali = Insan(2)
```

```
isinstance(ali, collections.abc.Iterator)
```

```
True
```

```
isinstance(ali, collections.abc.Iterable)
```

```
True
```

Now we can use *ali* in a for loop but,

```
# uncomment following two lines
# ali = Insan(2)
# for child in ali:
#     print(child)
```

but after the last iteration, we will get `IndexError` because of the way we have implemented the `__next__` method above.

Question: Elaborate the above mentioned reasoning?

```
class Insan:
    def __init__(self, num_child):
        self.children = [f"child_{i}" for i in range(num_child)]
        self.index = 0
```

(continues on next page)

(continued from previous page)

```
    def __next__(self):
        try:
            item = self.children[self.index]
        except IndexError:
            raise StopIteration

        self.index += 1
        return item

    def __iter__(self):
        return self


ali = Insan(2)
```

Above we have implemented the `__next__` method in a way to raise `StopIteration` error instead of `IndexError`. Since the `for` loop under the hood runs until `StopIteration` and then the for loop just bypasses the `StopIteration`, we can now use the *ali* in `for` loop safely.

```
for child in ali:
    print(child)
```

```
child_0
child_1
```

However, there is a problem in the above code, if we run the above for loop again, we don't get any output as shown below,

```
for child in ali:
    print(child)
```

This is because we are not not resetting `self.index` to 0 after raising `StopIteration` exception.

```
class Insan:
    def __init__(self, num_child):
        self.children = [f"child_{i}" for i in range(num_child)]
        self.index = 0

    def __next__(self):
        try:
            item = self.children[self.index]
        except IndexError:
            self.index = 0
            raise StopIteration

        self.index += 1
        return item

    def __iter__(self):
        return self
```

(continues on next page)

(continued from previous page)

```
ali = Insan(2)
for child in ali:
    print(child)
```

```
child_0
child_1
```

```
for child in ali:
    print(child)
```

```
child_0
child_1
```

__len__

This method determines the output of `len` function, when applied on the instance of a class.

```
class Family:
    def __init__(self, num_children):
        self.num_children = num_children

    def __len__(self):
        return 1 + 1 + self.num_children


fam = Family(3)

len(fam)
```

```
5
```

Since *fam* is instance of `Family` class, the answer to `len` function was same as we determined in `__len__` method.

Had we not defined the `__len__` method for *Family* class, we would have got `TypeError` if we had applied `len` function on it.

```
class Family:
    def __init__(self, num_children):
        self.num_children = num_children


fam = Family(3)

# uncomment the following line
# len(fam)  # -> TypeError: object of type 'Family' has no len()
```

`__getitem__` and `__setitem__`

If we define these methods for a class, then we can index the instance of the class using the slice operator i.e., `[]`.

```
class Data:
    def __init__(self, values):
        self.values = values

    def __getitem__(self, item):
        return self.values[item]

data = Data([1, 2, 3, 4])
```

```
print(data[0])
```

```
1
```

```
print(data[1])
```

```
2
```

```
# uncomment following two lines
# for idx in range(5):
#     print(data[idx])  # -> IndexError: list index out of range
```

The above example was too simple. Following example shows a more useful case for employment of `__getitem__` method where we would like to index two arrays simultaneously.

```
class Data:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __getitem__(self, item):
        return self.x[item], self.y[item]

data = Data([1,2,3], [11, 12, 13])
print(data[0])
```

```
(1, 11)
```

```
data = Data([1, 2, 3, 4], [11, 12, 13])

_x, _y = data[0]

print(_x, _y)
```

```
1 11
```

Even the lengths of x and y are not equal in above case, we were still able to slice them. We should have constructed the *Data* class in such a way to raise the error when the lengths are not equal. Without this, the error message becomes more confusing when the item is present in x but not in y.

```
# uncomment following line
# print(data[3])  # -> IndexError: list index out of range
```

```
class Data:
    def __init__(self, x, y):
        assert len(x) == len(y), 'length of x and y should be equal'
        self.x = x
        self.y = y
    def __getitem__(self, item):
        return self.x[item], self.y[item]
```

Now,if the lengths of x and y are not equal, we will get more useful error message.

```
# uncomment following line
# data = Data([1, 2, 3, 4], [11, 12, 13])  # -> AssertionError: length of x and y should␣
↪be equal
```

```
data = Data([1, 2, 3], [11, 12, 13])

_x, _y = data[0]

print(_x, _y)
```

```
1 11
```

__del__

This method determines what will happen to an object (instance of a class) when `del object` is executed.

```
class File:
    def __init__(self, name):
        self.path = os.path.join(os.getcwd(), name)
        with open(self.path, 'w'):
            pass
    def __del__(self):
        print(f"deleting file {self.path}")
        os.remove(self.path)
        return

f = File("test.txt")
```

```
os.path.exists(f.path)
```

```
True
```

```
del f
```

```
deleting file /home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/
↪latest/scripts/oop/test.txt
```

__contains__

This method determines what will happen when we use the instance of a class after `in` keyword.

```
class Country:
    def __init__(self, provinces:list):
        self.provinces = provinces
    def __contains__(self, item):
        return item in self.provinces
```

Country is a class which can have *provinces*.

```
pak = Country(['balochistan', 'kpk', 'sind', 'punjab', 'gb'])

print('sind' in pak)
```

```
True
```

```
print('sindh' in pak)
```

```
False
```

For a more comprehensive documentation on magical methods see this

Total running time of the script: (0 minutes 0.019 seconds)

3.17 getattr vs setattr

This lesson shows the usage of `setattr` and `getattr`

The `getattr` and `setattr` are builtin functions which are used to get and set an attribute to a python object. Following examples show how to use these functions to set and get attributes of a class.

setattr

```
class Human:

    def __init__(self, name):
        self.name = name

    def grow(self):
        setattr(self, 'empathy', 10)
        return


human = Human("Ali")
```

When we created the instance of *Human* class, it did not have *empathy* attribute.

```
# uncomment following line
# human.empathy  # -> AttributeError: 'Human' object has no attribute 'empathy'
```

After executing the *grow* method of *Human* class, the *empathy* attribute is set to it using `setattr` function.

```
human.grow()

print(human.empathy)
```

```
10
```

The first argument to `setattr` is the object to which we want to set the attribute. The second argument is the name of attribute and the third argument is the value of the attribute.

If we want to set/change the value of *empathy* attribute of *human* to 14, we can do this using `setattr` as below.

```
setattr(human, "empathy", 14)

print(human.empathy)
```

```
14
```

It will be obvious from above examples that the function `setattr` can be used both inside the class and outside the class definition.

```
human.empathy = 100

print(human.empathy)
```

```
100
```

From above example we can infer that doing *human.empathy = 10* is similar to *setattr(human, 'empathy', 100)*. This can be translated as, setting the attribute of *human* with the name *empathy* to 100.

getattr

`getattr` is opposite of `setattr`. It is used to fetch the attribute value of an object.

```
print(getattr(human, 'empathy'))
```

```
100
```

In other words, doing *human.empathy* is similar to running *getattr(human, 'empathy')* The second argument to both `setattr` and `getattr` is string (`str`) type.

```
class Human:

    def __init__(self, name):
        self.name = name

    def grow(self):
        setattr(self, 'empathy', 10)
        return

    def info(self):
```

(continues on next page)

(continued from previous page)

```
        empathy = getattr(self, 'empathy', None)
        return empathy


human = Human("Ali")
```

But what if the object does not have the attribute that we are trying to fetch?

```
# uncomment following line
# human.empathy  # -> AttributeError: 'Human' object has no attribute 'empathy'
```

Running the above code will give us AttributeError because, the *human* does not yet have *empathy* attribute.

```
human.info()
```

But why above cell did not throw AttributeError, even though we are getting, *empathy* attribute in it? This is because the 3rd argument in `getattr` function in *info* method is `None`. The 3rd argument is the default value of the attribute which we are trying to fetch. This means, when the object *human* did not have *empathy* attribute and we tried to get it using `getattr`, the default value `None` was returned.

We can verify this by printing the output of *human.info()*.

```
print(human.info())
```

```
None
```

However, if we run the *grow* method first, this will result in setting the *empathy* attribute to *human*. Consequently, we can see a different output when we run *human.info* after that.

```
human.grow()

print(human.info())
```

```
10
```

Question: What will be the output of following code?

```
class Man:
    def __init__(self, weight, height):
        self.bmi = weight / height**2

ali = Man(77, 1.71)
bmi = getattr(ali, 'bmi')
setattr(ali, 'bmi', bmi - 1.33288875)
print(ali.bmi)
```

Total running time of the script: (0 minutes 0.003 seconds)

2.7.4 4. numpy

Tutorials concerning numpy library .

4.1 understanding dimensions/axis

```
import numpy as np

print(np.__version__)
```

```
1.26.4
```

```
a = np.array([1.0])

print(a)
```

```
[1.]
```

```
print(a.shape)
```

```
(1,)
```

```
print(a.ndim)
```

```
1
```

```
a = np.array([1,2,3])

print(a)
```

```
[1 2 3]
```

```
print(f'length: {len(a)}, size: {a.size}, dim: {a.ndim}, shape: {a.shape}')
```

```
length: 3, size: 3, dim: 1, shape: (3,)
```

```
a = np.random.random((4, 5))

print(a)
```

```
[[0.35378575 0.84759879 0.41854462 0.91992659 0.74497   ]
 [0.01923873 0.79927105 0.05573417 0.0260903  0.03302782]
 [0.69334693 0.38664874 0.78804392 0.99024502 0.91512554]
 [0.40198701 0.52207574 0.99590104 0.03174812 0.41955784]]
```

```
print(f'length: {len(a)}, size: {a.size}, dim: {a.ndim}, shape: {a.shape}')
```

```
length: 4, size: 20, dim: 2, shape: (4, 5)
```

A numpy array should not be understood as rows and columns. Rather a numpy array of shape (4,5) can be understood as four arrays each of shape (5,)

```
a = np.random.random((3, 4, 5))

print(a)
```

```
[[[0.66568551 0.90116763 0.00801929 0.29054792 0.34485401]
  [0.3366532  0.23826046 0.05233746 0.89061753 0.84255441]
  [0.56661958 0.01119323 0.73329046 0.3381184  0.07716098]
  [0.15046486 0.50130673 0.96176383 0.05288099 0.65945255]]

 [[0.22441803 0.55969909 0.20168638 0.60704426 0.61939829]
  [0.17417084 0.54190168 0.9349729  0.51025176 0.30921539]
  [0.43000761 0.37355919 0.31378183 0.93033855 0.16454719]
  [0.35513501 0.60141758 0.6528549  0.97409538 0.47606376]]

 [[0.05991674 0.66851269 0.44354118 0.22650048 0.20250279]
  [0.84341428 0.44802401 0.22461202 0.58928467 0.2097547 ]
  [0.75339913 0.09154367 0.52222867 0.04363077 0.68649349]
  [0.52151979 0.7604434  0.81025123 0.33067682 0.23696642]]]
```

```
print(f'length: {len(a)}, size: {a.size}, dim: {a.ndim}, shape: {a.shape}')
```

```
length: 3, size: 60, dim: 3, shape: (3, 4, 5)
```

An array of shape (3, 4, 5) can be understood as three arrays each of shape (4, 5) while each of these (4, 5) arrays can be understood as four arrays each of shape (5,).

```
a = np.random.random((2, 3, 4, 5))

print(a)
```

```
[[[[1.69359765e-01 5.50819730e-01 5.41460474e-01 1.04482028e-01
    1.84001862e-01]
   [8.12001697e-01 9.13677827e-02 1.73072438e-01 1.27558768e-01
    5.61550461e-01]
   [7.93577028e-01 7.76283275e-01 6.50696796e-01 6.58061346e-01
    8.94507310e-01]
   [5.43754408e-01 5.19716278e-01 3.61511464e-01 5.26015488e-01
    1.08638888e-01]]

  [[7.26204666e-01 5.97159600e-01 2.63489205e-01 6.83840318e-02
    3.01083360e-01]
   [7.89346873e-01 6.39228312e-02 5.81418644e-01 2.45919527e-01
    2.55880484e-01]
   [1.00798631e-01 5.35423391e-01 4.96856892e-01 1.81955836e-01
    9.32971982e-01]
   [3.10477457e-01 4.89779092e-01 3.70458263e-01 9.82036740e-01
    8.07933340e-01]]
```

(continues on next page)

(continued from previous page)

```
 [[8.09800179e-01 5.68229451e-01 8.98368734e-01 1.56472554e-01
   1.31098535e-01]
  [6.70358618e-01 8.07093264e-01 2.15726554e-01 6.30372275e-01
   2.30381384e-01]
  [3.58972980e-01 3.48816486e-01 6.28577265e-01 3.44427240e-01
   7.89990271e-01]
  [7.17795110e-01 9.80600655e-01 8.28270907e-01 5.41211543e-01
   3.13045696e-01]]]


[[[5.13516067e-04 5.57521065e-01 5.43800772e-01 7.97971550e-01
   4.59060779e-01]
  [6.17071003e-01 6.00208461e-01 9.61822163e-01 6.60204191e-01
   4.05914058e-01]
  [7.41933818e-01 3.49233470e-01 3.81318754e-01 8.03641628e-01
   4.32943237e-01]
  [6.98233292e-01 2.70387837e-01 1.47348166e-01 7.46651779e-01
   6.57660205e-01]]

 [[5.88079775e-01 1.60354864e-01 4.43966351e-01 3.84860454e-01
   4.63659441e-01]
  [5.44312923e-01 1.99684175e-01 7.34406140e-01 9.60603254e-01
   1.19749957e-01]
  [3.08860456e-01 2.58157688e-01 5.15263756e-01 2.09102144e-01
   1.15740242e-01]
  [5.16025048e-01 7.94590735e-02 9.30896109e-01 3.41921896e-01
   5.88860746e-01]]

 [[7.56287507e-01 2.28018001e-01 1.43653648e-01 8.06950049e-01
   4.38242068e-02]
  [9.61808835e-01 2.46091983e-01 7.42262312e-01 8.95480792e-01
   2.35714429e-01]
  [9.01857536e-01 9.29747970e-01 6.54127973e-01 5.03798662e-01
   4.28908478e-01]
  [7.14703348e-01 3.68232939e-01 5.84643438e-01 1.43750937e-01
   3.41964591e-01]]]]
```

```
print(f'length: {len(a)}, size: {a.size}, dim: {a.ndim}, shape: {a.shape}')
```

```
length: 2, size: 120, dim: 4, shape: (2, 3, 4, 5)
```

Similarly an array of shape (2,3,4,5) can be understood as two arrays each of shape (3,4,5) and so on.

Thinking in this way, makes it easier to conceptualize higher dimensional arrays.

The naming of axis in numpy is also consistent with this understanding. The axis starts from 0 and goes up to n-1 where n is the number of dimensions. The axis 0 is the outermost axis and the axis n-1 is the innermost axis. Consider the example of mean. If we do np.mean(a), it will calculate the mean of all the elements in the array.

```
print(np.mean(a))
```

```
0.49290649748986193
```

If we do np.mean(a, axis=0), it will calculate the mean along the axis 0.

```
print(np.mean(a, axis=0).shape)
```

```
(3, 4, 5)
```

If we do np.mean(a, axis=1), it will calculate the mean along the axis 1.

```
print(np.mean(a, axis=1).shape)
```

```
(2, 4, 5)
```

If we do np.mean(a, axis=2), it will calculate the mean along the axis 2.

```
print(np.mean(a, axis=2).shape)
```

```
(2, 3, 5)
```

If we do np.mean(a, axis=3), it will calculate the mean along the axis 3.

```
print(np.mean(a, axis=3).shape)
```

```
(2, 3, 4)
```

But if we specify an axis that does not exist, it will give an `AxisError` error.

```
# Uncomment the following line to see the error
#
# print(np.mean(a, axis=4).shape)
```

Total running time of the script: (0 minutes 0.010 seconds)

4.2 stacking vs concatenating

This lesson illustrates difference between `stack`, `vstack`, `hstack`, `column_stack`, `row_stack` and `concatenate`

```
import time
import numpy as np

print(time.asctime())
print(np.__version__)
```

```
Mon Nov 11 19:32:23 2024
1.26.4
```

stack

all arrays must have same shape

both 1d arrays

```
a = np.random.random(10)
b = np.random.random(10)

print(np.stack([a,b]).shape)
```

```
(2, 10)
```

both 2D

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.stack([a,b]).shape)
```

```
(2, 10, 1)
```

```
print(np.stack([a,b], axis=1).shape)
```

```
(10, 2, 1)
```

```
print(np.stack([a,b], axis=2).shape)
```

```
(10, 1, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.stack([a,b]).shape)
```

```
(2, 10, 2)
```

```
print(np.stack([a,b], axis=0).shape)
```

```
(2, 10, 2)
```

```
print(np.stack([a,b], axis=1).shape)
```

```
(10, 2, 2)
```

```
print(np.stack([a,b], axis=2).shape)
```

```
(10, 2, 2)
```

```
# print(np.stack([a,b], axis=3).shape) # np.AxisError
```

different shapes

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

# print(np.stack([a,b]).shape)  # ValueError
```

```
# print(np.stack([a,b], axis=0).shape)  # ValueError
```

```
# print(np.stack([a,b], axis=1).shape)  # ValueError
```

concatenate

```
a = np.random.random(10)
b = np.random.random(10)

print(np.concatenate([a,b]).shape)
```

```
(20,)
```

```
print(np.concatenate([a,b], axis=0).shape)
```

```
(20,)
```

```
# print(np.concatenate([a,b], axis=1).shape)  # Error
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.concatenate([a,b]).shape)
```

```
(20, 1)
```

```
print(np.concatenate([a,b], axis=1).shape)
```

```
(10, 2)
```

The shapes of the arrays must be same except in the dimension corresponding to axis

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

# print(np.concatenate([a,b], axis=0)) # Error
```

In above example, axis 0 has 10 but axis 1 has 2 and 1. So, it is not possible to concatenate

```
print(np.concatenate([a,b], axis=1).shape)
```

```
(10, 3)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.concatenate([a,b]).shape)
```

```
(20, 2)
```

```
print(np.concatenate([a,b], axis=1).shape)
```

```
(10, 4)
```

```
# print(np.concatenate([a,b], axis=2).shape)  # AxisError
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.concatenate([a,b]).shape)
```

```
(20, 5, 3)
```

```
print(np.concatenate([a,b], axis=1).shape)
```

```
(10, 10, 3)
```

```
print(np.concatenate([a,b], axis=2).shape)
```

```
(10, 5, 6)
```

vstack

```
a = np.random.random(10)
b = np.random.random(10)

print(np.vstack([a,b]).shape)
```

```
(2, 10)
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.vstack([a,b]).shape)
```

```
(20, 1)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.vstack([a,b]).shape)
```

```
(20, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

print(np.hstack([a,b]).shape)
```

```
(10, 3)
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.hstack([a,b]).shape)
```

```
(10, 10, 3)
```

hstack

```
a = np.random.random(10)
b = np.random.random(10)

print(np.hstack([a,b]).shape)
```

```
(20,)
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.hstack([a,b]).shape)
```

```
(10, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.hstack([a,b]).shape)
```

```
(10, 4)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

print(np.hstack([a,b]).shape)
```

```
(10, 3)
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.hstack([a,b]).shape)
```

```
(10, 10, 3)
```

column stack

```
a = np.random.random(10)
b = np.random.random(10)

print(np.column_stack([a,b]).shape)
```

```
(10, 2)
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.column_stack([a,b]).shape)
```

```
(10, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.column_stack([a,b]).shape)
```

```
(10, 4)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

print(np.column_stack([a,b]).shape)
```

```
(10, 3)
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.column_stack([a,b]).shape)
```

```
(10, 10, 3)
```

row stack

```
a = np.random.random(10)
b = np.random.random(10)

print(np.row_stack([a,b]).shape)
```

```
(2, 10)
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.row_stack([a,b]).shape)
```

```
(20, 1)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.row_stack([a,b]).shape)
```

```
(20, 2)
```

shape has to be same

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

# print(np.row_stack([a,b]).shape) ValueError
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.row_stack([a,b]).shape)
```

```
(20, 5, 3)
```

dstack

depth wise stacking

```
a = np.random.random(10)
b = np.random.random(10)

print(np.dstack([a,b]).shape)
```

```
(1, 10, 2)
```

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print(np.dstack([a,b]).shape)
```

```
(10, 1, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print(np.dstack([a,b]).shape)
```

```
(10, 2, 2)
```

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

# print(np.dstack([a,b]).shape)  # ValueError
```

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print(np.dstack([a,b]).shape)
```

```
(10, 5, 6)
```

comparison

1D arrays

```
a = np.random.random(10)
b = np.random.random(10)

print('concatenate: ', np.concatenate([a,b]).shape)

print('stack        ', np.stack([a,b]).shape)

print('vstack:      ', np.vstack([a,b]).shape)

print('hstack:      ', np.hstack([a,b]).shape)

print('row_stack:   ', np.row_stack([a,b]).shape)

print('column_stack:', np.column_stack([a,b]).shape)

print('dstack:      ', np.dstack([a,b]).shape)
```

```
concatenate:  (20,)
stack         (2, 10)
vstack:       (2, 10)
hstack:       (20,)
row_stack:    (2, 10)
column_stack: (10, 2)
dstack:       (1, 10, 2)
```

2D arrays

```
a = np.random.random((10, 1))
b = np.random.random((10, 1))

print('concatenate:  ', np.concatenate([a,b]).shape)

print('stack:        ', np.stack([a,b]).shape)

print('vstack:       ', np.vstack([a,b]).shape)

print('hstack:       ', np.hstack([a,b]).shape)

print('row_stack:    ', np.row_stack([a,b]).shape)

print('column_stack: ', np.column_stack([a,b]).shape)

print('dstack:       ', np.dstack([a,b]).shape)
```

```
concatenate:   (20, 1)
stack:         (2, 10, 1)
vstack:        (20, 1)
hstack:        (10, 2)
row_stack:     (20, 1)
column_stack:  (10, 2)
dstack:        (10, 1, 2)
```

2D arrays

```
a = np.random.random((10, 2))
b = np.random.random((10, 2))

print('concatenate: ', np.concatenate([a,b]).shape)

print('stack:       ', np.stack([a,b]).shape)

print('vstack:      ', np.vstack([a,b]).shape)

print('hstack:      ', np.hstack([a,b]).shape)

print('row_stack:   ', np.row_stack([a,b]).shape)

print('column_stack:',np.column_stack([a,b]).shape)

print('dstack:      ', np.dstack([a,b]).shape)
```

```
concatenate:  (20, 2)
stack:        (2, 10, 2)
vstack:       (20, 2)
hstack:       (10, 4)
row_stack:    (20, 2)
column_stack: (10, 4)
dstack:       (10, 2, 2)
```

2D arrays with different shapes

```
a = np.random.random((10, 2))
b = np.random.random((10, 1))

# print(np.concatenate([a,b]).shape)  # ValueError

# print(np.stack([a,b]).shape)  # ValueError

# print(np.vstack([a,b]).shape) # ValueError

print('hstack:       ', np.hstack([a,b]).shape)

# print(np.row_stack([a,b]).shape) # ValueError

print('column_stack: ', np.column_stack([a,b]).shape)

# print(np.dstack([a,b]).shape)  # ValueError
```

```
hstack:        (10, 3)
column_stack:  (10, 3)
```

3D arrays

```
a = np.random.random((10, 5, 3))
b = np.random.random((10, 5, 3))

print('concatenate: ', np.concatenate([a,b]).shape)

print('stack:       ', np.stack([a,b]).shape)

print('vstack:      ', np.vstack([a,b]).shape)

print('hstack:      ', np.hstack([a,b]).shape)

print('row_stack:   ', np.row_stack([a,b]).shape)

print('column_stack:', np.column_stack([a,b]).shape)

print('dstack:      ', np.dstack([a,b]).shape)
```

```
concatenate:  (20, 5, 3)
stack:        (2, 10, 5, 3)
vstack:       (20, 5, 3)
hstack:       (10, 10, 3)
```

(continues on next page)

(continued from previous page)

```
row_stack:    (20, 5, 3)
column_stack: (10, 10, 3)
dstack:       (10, 5, 6)
```

Total running time of the script: (0 minutes 0.025 seconds)

4.3 quantile and digitize

This file describes the concept of quantile and how to cclculate it in numpy and various functions around it.

```
import numpy as np

print(np.__version__)
```

```
1.26.4
```

```
x = np.array([1,2,3,4,5])

np.quantile(x, 0.5)
```

```
3.0
```

```
x = np.array([1,2,2,3,3,4,5])

np.quantile(x, 0.5)
```

```
3.0
```

so quantile actually means that what will be that value that if we distribute the values of the array 50% on one side and 50% on other side. 50% because we have used 0.5

```
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

print(np.quantile(x, 0.5))
```

```
4.5
```

```
print(np.quantile(x, 0.2))
```

```
1.8
```

```
print(np.quantile(x, [0.2, 0.5]))
```

```
[1.8 4.5]
```

```
print(np.quantile(np.arange(10, 20), [0.2, 0.5]))
```

```
[11.8 14.5]
```

what if the array is 2D

```
print(np.quantile(np.arange(20).reshape(-1, 10), [0.2, 0.5]))
```

```
[3.8 9.5]
```

```
np.quantile([1.8, 11.8], 0.2)
# 3.8
np.quantile([4.5, 14.5], 0.5)
# 9.5
```

Total running time of the script: (0 minutes 0.005 seconds)

2.7.5 5. pandas

Tutorials concerning pandas library

5.1 introduction

```
import time
import numpy as np
import pandas as pd

print(time.asctime())
print(pd.__version__, np.__version__)
```

```
Mon Nov 11 19:32:24 2024
1.5.3 1.26.4
```

Suppose we have an array [0.4, 0.3, 0.5, 0.2, 0.6, 0.3]. Let's say the values in this array represent concentrations in water measured every hour from 13 pm to 19 pm. However, with just an array, we don't have the ability to encode this information. If we want to add the (temporal) reference of each value we have to add it ourself for example by saving that in a separate array. Pandas comes with this in-built ability that we can add reference or labels to arrays. Every array in pandas has two kinds of references. The reference for the rows which is called `index` and the reference for the columns which is called `columns`. Therefore we can call pandas a library which have referenced/labelled arrays.

The core data structure in pandas is `DataFrame` which consists of one or more columns. A single column in a DataFrame is a `Series`.

```
df = pd.DataFrame(np.random.random((10, 3)))
print(df)
```

```
          0         1         2
0  0.819711  0.469719  0.047138
1  0.995136  0.799338  0.014367
2  0.220825  0.169214  0.077614
3  0.472161  0.297704  0.213042
4  0.096419  0.898009  0.074485
5  0.210453  0.086511  0.951945
6  0.461789  0.427172  0.143734
7  0.906187  0.200208  0.497694
```

(continues on next page)

(continued from previous page)

```
8  0.477416  0.382160  0.434236
9  0.428262  0.910327  0.273702
```

The data in columns is stored as numpy arrays. Therefore, a DataFrames and Series have a lot of characteristics similar to that of numpy arrays.

```
print(df.shape)
```

```
(10, 3)
```

By default the columns names are just integers starting from 0, however we can define the column names ourselves as well.

```
df = pd.DataFrame(np.random.random((10, 3)), columns=['a', 'b', 'c'])
print(df)
```

```
          a         b         c
0  0.643263  0.594007  0.442692
1  0.286817  0.179552  0.060201
2  0.463696  0.816541  0.600126
3  0.150433  0.366931  0.906603
4  0.288914  0.052968  0.615138
5  0.158648  0.694414  0.973317
6  0.744459  0.436719  0.882664
7  0.175492  0.733806  0.173178
8  0.051456  0.641243  0.885918
9  0.474923  0.216120  0.883705
```

```
print(df.columns)
```

```
Index(['a', 'b', 'c'], dtype='object')
```

The columns are list like structures. However they are not exactly lists.

```
print(type(df.columns))
```

```
<class 'pandas.core.indexes.base.Index'>
```

We can however, convert the columns to list though.

```
df.columns.to_list()
```

```
['a', 'b', 'c']
```

```
print(type(df.columns.to_list()))
```

```
<class 'list'>
```

The default label for the rows i.e. `index` consists of numbers starting from 0.

```
print(df.index)
```

```
RangeIndex(start=0, stop=10, step=1)
```

However, we can set `index` of our choice as well.

```
df = pd.DataFrame(np.random.random((10, 3)),
                  columns=['a', 'b', 'c'],
                 index=[2000+i for i in range(10)])
print(df)
```

```
             a         b         c
2000  0.978551  0.820233  0.227459
2001  0.317172  0.157027  0.695845
2002  0.767413  0.547235  0.369868
2003  0.425290  0.032933  0.923786
2004  0.251980  0.580116  0.266811
2005  0.947292  0.581080  0.211948
2006  0.868994  0.163130  0.771538
2007  0.627722  0.624592  0.296365
2008  0.337227  0.355143  0.516186
2009  0.441642  0.761122  0.799617
```

```
print(df.index)
```

```
Int64Index([2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009], dtype='int64')
```

The default name of `index` is `None`.

```
print(df.index.name)
```

```
None
```

However, we can set the name of index as well.

```
df.index.name = 'years'
print(df)
```

```
             a         b         c
years
2000  0.978551  0.820233  0.227459
2001  0.317172  0.157027  0.695845
2002  0.767413  0.547235  0.369868
2003  0.425290  0.032933  0.923786
2004  0.251980  0.580116  0.266811
2005  0.947292  0.581080  0.211948
2006  0.868994  0.163130  0.771538
2007  0.627722  0.624592  0.296365
2008  0.337227  0.355143  0.516186
2009  0.441642  0.761122  0.799617
```

```
print(df.index.name)
```

```
years
```

```
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
df = pd.DataFrame(np.random.randint(0, 10, (10, 1)),
                  columns=['a'],
                 index=[2000+i for i in range(10)])
print(df)
```

```
      a
2000  1
2001  6
2002  8
2003  5
2004  0
2005  6
2006  2
2007  8
2008  2
2009  3
```

```
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
print(df.columns)
```

```
Index(['a'], dtype='object')
```

Series

A Series consists of a single column. It can be constructed using `pd.Series`.

```
s = pd.Series(np.random.random(10))
print(s)
```

```
0    0.240092
1    0.316543
2    0.336670
3    0.256285
4    0.662836
5    0.947630
6    0.188108
7    0.525331
```

(continues on next page)

(continued from previous page)

```
8    0.612323
9    0.212473
dtype: float64
```

```
print(type(s))
```

```
<class 'pandas.core.series.Series'>
```

```
print(s.shape)
```

```
(10,)
```

```
print(s.name)
```

```
None
```

```
s = pd.Series(np.random.random(10),
              name="a")
print(s)
```

```
0    0.342054
1    0.232343
2    0.547047
3    0.942240
4    0.766106
5    0.313809
6    0.783459
7    0.325427
8    0.380389
9    0.418716
Name: a, dtype: float64
```

```
print(s.name)
```

```
a
```

the Series is literally the data structure for a single column of a DataFrame.

```
df = pd.DataFrame(np.random.random((10, 3)),
                  columns=['a', 'b', 'c'],
                 index=[2000+i for i in range(10)])
print(df)
```

```
             a         b         c
2000  0.611481  0.646587  0.884365
2001  0.884323  0.265456  0.499323
2002  0.035322  0.413177  0.404945
2003  0.138943  0.159374  0.096236
2004  0.007718  0.173929  0.156492
```

(continues on next page)

(continued from previous page)

```
2005  0.244785  0.159650  0.829702
2006  0.226925  0.632816  0.999603
2007  0.251388  0.524466  0.769429
2008  0.024436  0.454759  0.473400
2009  0.611078  0.184098  0.176071
```

A single column in a DataFrame is a Series.

```
print(type(df['a']))
```

```
<class 'pandas.core.series.Series'>
```

```
s = pd.Series(np.random.random(10),
              index=[2000+i for i in range(10)],
              name="a")
print(s)
```

```
2000    0.520191
2001    0.355112
2002    0.645158
2003    0.384966
2004    0.996232
2005    0.068262
2006    0.720012
2007    0.611447
2008    0.931790
2009    0.001561
Name: a, dtype: float64
```

Since pandas is based upon numpy arrays. We can extract actual numpy arrays from DataFrame using *.values* method.

```
print(df.values)
```

```
[[0.61148101 0.64658729 0.8843646 ]
 [0.88432322 0.26545644 0.499323  ]
 [0.03532233 0.41317685 0.40494501]
 [0.13894266 0.15937354 0.0962357 ]
 [0.00771782 0.17392933 0.15649219]
 [0.24478489 0.15965015 0.82970221]
 [0.22692541 0.63281557 0.99960347]
 [0.25138839 0.52446576 0.76942873]
 [0.0244364  0.4547589  0.47339962]
 [0.61107766 0.1840978  0.17607135]]
```

```
print(type(df.values))
```

```
<class 'numpy.ndarray'>
```

```
df = pd.DataFrame(np.random.randint(0, 14, (10, 3)),
                  columns=['a', 'b', 'c'],
```

(continues on next page)

(continued from previous page)

```
                  index=[2000+i for i in range(10)])
print(df)
```

```
      a   b   c
2000  12  10   1
2001   6   5   2
2002   2   0   4
2003  12   1  12
2004  11   1   0
2005   1   9  13
2006   6   3  11
2007   9   0  11
2008   7   7   0
2009   5   8   2
```

```
print(type(df.values))
```

```
<class 'numpy.ndarray'>
```

```
print(df.values.shape)
```

```
(10, 3)
```

```
df.describe()
```

```
df.head()
```

```
df.head(8)
```

Get the last N rows of a DataFrame

```
df.tail()
```

```
df.tail(7)
```

```
df.mean()
```

```
a    7.1
b    4.4
c    5.6
dtype: float64
```

```
df.to_dict()
```

```
{'a': {2000: 12, 2001: 6, 2002: 2, 2003: 12, 2004: 11, 2005: 1, 2006: 6, 2007: 9, 2008:␣
↪7, 2009: 5}, 'b': {2000: 10, 2001: 5, 2002: 0, 2003: 1, 2004: 1, 2005: 9, 2006: 3,␣
↪2007: 0, 2008: 7, 2009: 8}, 'c': {2000: 1, 2001: 2, 2002: 4, 2003: 12, 2004: 0, 2005:␣
↪13, 2006: 11, 2007: 11, 2008: 0, 2009: 2}}
```

```
df.to_dict('list')
```

```
{'a': [12, 6, 2, 12, 11, 1, 6, 9, 7, 5], 'b': [10, 5, 0, 1, 1, 9, 3, 0, 7, 8], 'c': [1,␣
↪2, 4, 12, 0, 13, 11, 11, 0, 2]}
```

```
df['d'] = np.random.randint(0, 10, (10,))
print(df)
```

```
      a   b   c  d
2000  12  10   1  4
2001   6   5   2  4
2002   2   0   4  6
2003  12   1  12  9
2004  11   1   0  5
2005   1   9  13  5
2006   6   3  11  5
2007   9   0  11  5
2008   7   7   0  0
2009   5   8   2  8
```

```
df.pop('d')
print(df)
```

```
      a   b   c
2000  12  10   1
2001   6   5   2
2002   2   0   4
2003  12   1  12
2004  11   1   0
2005   1   9  13
2006   6   3  11
2007   9   0  11
2008   7   7   0
2009   5   8   2
```

```
df.columns = ['x', 'y', 'z']
print(df)
```

```
      x   y   z
2000  12  10   1
2001   6   5   2
2002   2   0   4
2003  12   1  12
2004  11   1   0
2005   1   9  13
2006   6   3  11
2007   9   0  11
2008   7   7   0
2009   5   8   2
```

row count of pandas dataframe

```
len(df.index)
```

```
10
```

```
print(df.shape[0])
```

```
10
```

change the order of DataFrame columns

```
cols = df.columns.tolist()
cols = cols[-1:] + cols[:-1]
df = df[cols]
print(df)
```

```
      z   x   y
2000  1  12  10
2001  2   6   5
2002  4   2   0
2003 12  12   1
2004  0  11   1
2005 13   1   9
2006 11   6   3
2007 11   9   0
2008  0   7   7
2009  2   5   8
```

drop rows of Pandas DataFrame whose value in a certain column is NaN

```
df = pd.DataFrame(np.random.randn(6,3))
print(df)
```

```
          0         1         2
0  0.792738 -2.566105  0.434209
1  1.059024 -0.031717  0.358016
2  0.301728  0.249756 -0.682732
3 -0.944110  1.394479  1.437554
4  1.052420  1.079581  0.149270
5 -1.030245 -0.500275 -2.613331
```

```
df.iloc[::2,0] = np.nan; df.iloc[::4,2] = np.nan; df.iloc[::3,2] = np.nan
print(df)
```

```
          0         1         2
0       NaN -2.566105       NaN
1  1.059024 -0.031717  0.358016
2       NaN  0.249756 -0.682732
3 -0.944110  1.394479       NaN
4       NaN  1.079581       NaN
5 -1.030245 -0.500275 -2.613331
```

dropping all rows having NaN values

```
df.dropna()
```

dropping NaN in specific columns

```
print(df[df[2].notna()])
```

```
          0         1         2
1  1.059024 -0.031717  0.358016
2       NaN  0.249756 -0.682732
5 -1.030245 -0.500275 -2.613331
```

count the NaN values in a column in DataFrame

```
df = pd.DataFrame(np.random.randn(6,3))
df.iloc[::2,0] = np.nan; df.iloc[::4,2] = np.nan; df.iloc[::3,2] = np.nan
print(df)
```

```
          0         1         2
0       NaN  0.139581       NaN
1 -1.509370  1.762801 -1.067253
2       NaN  0.095276 -0.777572
3  0.492278  0.061235       NaN
4       NaN -0.038476       NaN
5 -0.067574 -0.885610 -0.721969
```

```
df.isna().sum()
```

```
0    3
1    0
2    3
dtype: int64
```

for columns

```
df.isnull().sum(axis = 0)
```

```
0    3
1    0
2    3
dtype: int64
```

for rows

```
df.isnull().sum(axis = 1)
```

```
0    2
1    0
2    1
3    1
4    2
5    0
dtype: int64
```

check if any value is NaN in a DataFrame

```
df = pd.DataFrame(np.random.randn(6,3))
df.iloc[::2,0] = np.nan; df.iloc[::4,2] = np.nan; df.iloc[::3,2] = np.nan
print(df)
```

```
          0         1         2
0       NaN  0.163625       NaN
1 -1.087252  2.721640 -0.415199
2       NaN  1.432081 -0.653284
3  1.336880  0.197294       NaN
4       NaN  0.015589       NaN
5  1.388623  0.203641  1.691277
```

how many NaN

```
df.isnull()
```

column wise

```
df.isnull().any()
```

```
0     True
1    False
2     True
dtype: bool
```

if there is any NaN in entire data

```
df.isnull().any().any()
```

```
True
```

replace NaN values by Zeroes in a column of a Dataframe?

```
df = pd.DataFrame(np.random.randn(6,3))
df.iloc[::2,0] = np.nan; df.iloc[::4,2] = np.nan; df.iloc[::3,2] = np.nan
print(df)
```

```
          0         1         2
0       NaN -0.168681       NaN
1  0.563927  0.017890 -1.375824
2       NaN -0.810597 -1.174800
3 -0.722840  0.883346       NaN
4       NaN -0.310959       NaN
5  0.940451 -1.261573 -1.127101
```

```
df.fillna(0)
```

To fill the NaNs in only one column

```
df[2].fillna(0, inplace=True)
print(df)
```

```
          0         1         2
0       NaN -0.168681  0.000000
1  0.563927  0.017890 -1.375824
2       NaN -0.810597 -1.174800
3 -0.722840  0.883346  0.000000
4       NaN -0.310959  0.000000
5  0.940451 -1.261573 -1.127101
```

check if a column exists in Pandas

```
df = pd.DataFrame(np.random.randn(6,3))
print(df)
```

```
          0         1         2
0  0.775496  0.028101  0.380929
1 -0.932216  0.469528 -0.663859
2 -0.616558 -1.267830  0.580904
3 -1.582028 -0.355916  0.460871
4  0.672658 -1.117510  0.144625
5  0.613005  0.732261 -0.276066
```

```
if 0 in df.columns:
    print("true")
```

```
true
```

Python dict into a dataframe

```
d = {
    '2012-06-08': 388,
    '2012-06-09': 388,
    '2012-06-10': 388,
    '2012-06-11': 389,
    '2012-06-12': 389,
    '2012-06-13': 389,
    '2012-06-14': 389,
    '2012-06-15': 389,
    '2012-06-16': 389,
    '2012-06-17': 389,
    '2012-06-18': 390,
    '2012-06-19': 390,
    '2012-06-20': 390,
}
```

```
pd.DataFrame(d.items())
```

```
pd.DataFrame(d.items(), columns=['Date', 'DateValue'])
```

uncomment following line pd.DataFrame(d) # ValueError: If using all scalar values, you must pass an index

```
pd.DataFrame([d])
```

```
pd.DataFrame.from_dict(d, orient='index', columns=['DateVaue'])
```

Count the frequency that a value occurs in a dataframe column

```
df = pd.DataFrame(np.random.randint(0, 14, (10, 3)),
                  columns=['a', 'b', 'c'],
                 index=[2000+i for i in range(10)])
df['a'].value_counts()
```

```
6     2
2     2
12    2
9     1
11    1
0     1
5     1
Name: a, dtype: int64
```

```
for index, row in df.iterrows():
    print(index, row, '\n')
```

```
2000 a     6
b    12
c     5
Name: 2000, dtype: int64

2001 a     2
b     1
c    10
Name: 2001, dtype: int64

2002 a     6
b     0
c    12
Name: 2002, dtype: int64

2003 a    9
b    8
c    7
Name: 2003, dtype: int64

2004 a    11
b    11
c     4
Name: 2004, dtype: int64

2005 a    0
b    0
c    7
Name: 2005, dtype: int64

2006 a    12
```

(continues on next page)

(continued from previous page)

```
b     6
c     8
Name: 2006, dtype: int64

2007 a    12
b     5
c     1
Name: 2007, dtype: int64

2008 a     2
b     2
c    13
Name: 2008, dtype: int64

2009 a     5
b     2
c    13
Name: 2009, dtype: int64
```

```
df = pd.DataFrame(np.random.randint(0, 14, (10, 3)),
                  columns=['a', 'b', 'c'])
print(df)
```

```
    a   b   c
0   9  10  13
1   7   7   3
2   5   2  13
3   8   8   7
4   7   6   3
5   3   1   5
6   4  10   7
7   0   8   5
8  12  12  12
9  13   4   9
```

```
print(df['a']/df['b'])
```

```
0    0.900000
1    1.000000
2    2.500000
3    1.000000
4    1.166667
5    3.000000
6    0.400000
7    0.000000
8    1.000000
9    3.250000
dtype: float64
```

add an empty column to a dataframe?

```
df["d"] = ""
print(df)
```

```
    a   b   c d
0   9  10  13
1   7   7   3
2   5   2  13
3   8   8   7
4   7   6   3
5   3   1   5
6   4  10   7
7   0   8   5
8  12  12  12
9  13   4   9
```

```
print(df['d'])
```

```
0
1
2
3
4
5
6
7
8
9
Name: d, dtype: object
```

```
df["d"] = np.nan
print(df)
```

```
    a   b   c   d
0   9  10  13 NaN
1   7   7   3 NaN
2   5   2  13 NaN
3   8   8   7 NaN
4   7   6   3 NaN
5   3   1   5 NaN
6   4  10   7 NaN
7   0   8   5 NaN
8  12  12  12 NaN
9  13   4   9 NaN
```

What does axis in pandas mean?

```
df.mean(axis=0)
```

```
a    6.8
b    6.8
c    7.7
```

(continues on next page)

(continued from previous page)

```
d    NaN
dtype: float64
```

```
df.mean(axis=1)
```

```
0    10.666667
1     5.666667
2     6.666667
3     7.666667
4     5.333333
5     3.000000
6     7.000000
7     4.333333
8    12.000000
9     8.666667
dtype: float64
```

Replace NaN with blank/empty string

```
df.replace(9, np.nan)
```

```
df.replace(np.nan, '')
```

Rename specific column(s) in pandas

```
df = pd.DataFrame(np.random.randint(0, 14, (10, 3)), columns=['a', 'b', 'c'])
print(df)
```

```
    a   b   c
0   2   4  12
1   9  13   2
2  11   1   4
3   9   9   5
4   9   5   4
5  12   0   1
6  10   2  11
7   6   6   1
8  12   7   4
9   6   9   7
```

```
df.rename(columns={'a':'log(A)'}, inplace=True)
print(df)
```

```
   log(A)   b   c
0       2   4  12
1       9  13   2
2      11   1   4
3       9   9   5
4       9   5   4
5      12   0   1
6      10   2  11
```

(continues on next page)

(continued from previous page)

```
7       6   6   1
8      12   7   4
9       6   9   7
```

print DataFrame without index

```
print(df)
```

```
   log(A)   b   c
0       2   4  12
1       9  13   2
2      11   1   4
3       9   9   5
4       9   5   4
5      12   0   1
6      10   2  11
7       6   6   1
8      12   7   4
9       6   9   7
```

```
df.style.hide_index()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪pandas/dataframe_vs_series.py:478: FutureWarning: this method is deprecated in favour␣
↪of `Styler.hide(axis="index")`
  df.style.hide_index()
```

replace nan values with average of columns

```
df.fillna(df.mean())
```

retrieve the number of columns in a dataframe?

```
len(df.columns)
```

```
3
```

```
print(df.shape[1])
```

```
3
```

We can create empty DataFrame by telling how many columns should exist or how many rows should exist.

```
df = pd.DataFrame(columns=['A','B','C','D','E','F','G'])
print(df)
```

```
Empty DataFrame
Columns: [A, B, C, D, E, F, G]
Index: []
```

```
print(df.shape)
```

```
(0, 7)
```

```
df = pd.DataFrame(index=range(1,8))
print(df)
```

```
Empty DataFrame
Columns: []
Index: [1, 2, 3, 4, 5, 6, 7]
```

```
print(df.shape)
```

```
(7, 0)
```

Total running time of the script: (0 minutes 0.461 seconds)

5.2 indexing and slicing

This lesson shows how to select rows or columns from pandas dataframe.

```
import pandas as pd
print(pd.__version__)
```

```
1.5.3
```

```
pd.set_option('display.max_columns', 11)
```

Let's create a dataframe

```
df = pd.DataFrame({
    'name':  ['Ali', 'Hasan', 'Husain', 'Ali', 'Muhammad', 'Jafar', 'Musa', 'Raza', 'Taqi
↪', 'Naqi', 'Askari'],
    'age':   [63,     47,      57,       57,     57,         65,      55,     55,    24,␣
↪    40,     27],
    'other': ['muavia','muavia','yazid','walid','hisham','mansur','harun','mamun',
↪'Mutasim','Mutasim','mutaz'],
    'cityb': ['MKH',    'MAD',   'MAD',  'MAD',  'MAD',   'MAD',   'MAD',  'MAD',  'MAD',
↪    'MAD',   'MAD'],
    'duration':[29,      10,      11,       34,     19,    32,      35,     20,     16, ␣
↪    34,       6],
    'YoB_H': [-22,       3,       4,        38,     56,    83,      128,    148,    195,␣
↪    212,      232],
    'YOB_G': [600,       625,     626,      659,    676,   702,     745,    766,    811,␣
↪    828,      844],
    'YoM_H': [40,        50,      61,       95,     114,   148,     183,    203,    220,␣
↪    254,      260],
    'YoM_G': [661,       670,     680,      712,    733,   765,     799,    818,    835,␣
↪    868,      874],
    'dynasty':[None,'umayad','umayad','umayad','umayad','abbasid','abbasid','abbasid',
↪'abbasid','abbasid','abbasid'],
```

(continues on next page)

(continued from previous page)

```
    'cityd': ['NJF', 'MAD', 'KBL', 'MAD', 'MAD', 'MAD', 'BGD', 'MAS',
↪'BGH', 'SAM', 'SAM']
},
    index=['first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh', 'eigth',
↪'ninth', 'tenth', 'eleventh']
)
print(df)
```

```
             name  age    other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
first         Ali   63   muavia   MKH        29    -22    600     40    661
second      Hasan   47   muavia   MAD        10      3    625     50    670
third      Husain   57    yazid   MAD        11      4    626     61    680
fourth        Ali   57    walid   MAD        34     38    659     95    712
fifth    Muhammad   57   hisham   MAD        19     56    676    114    733
sixth       Jafar   65   mansur   MAD        32     83    702    148    765
seventh      Musa   55    harun   MAD        35    128    745    183    799
eigth        Raza   55    mamun   MAD        20    148    766    203    818
ninth        Taqi   24  Mutasim   MAD        16    195    811    220    835
tenth        Naqi   40  Mutasim   MAD        34    212    828    254    868
eleventh   Askari   27    mutaz   MAD         6    232    844    260    874

          dynasty cityd
first        None   NJF
second     umayad   MAD
third      umayad   KBL
fourth     umayad   MAD
fifth      umayad   MAD
sixth     abbasid   MAD
seventh   abbasid   BGD
eigth     abbasid   MAS
ninth     abbasid   BGH
tenth     abbasid   SAM
eleventh  abbasid   SAM
```

```
print(df.shape)
```

```
(11, 11)
```

The indexing operator, `[]`, can be used for slicing and for selecting rows and columns. However, it can not be used for both purposes (slicing and selecting rows/columns) simultaneously.

We can select a single column from dataframe as below

```
print(df['name'])
```

```
first           Ali
second        Hasan
third        Husain
fourth          Ali
fifth      Muhammad
sixth         Jafar
seventh        Musa
```

(continues on next page)

(continued from previous page)

```
eigth           Raza
ninth           Taqi
tenth           Naqi
eleventh      Askari
Name: name, dtype: object
```

For selecting multiple columns, we must pass a list of columns.

```
print(df[['other', 'YoM_H']])
```

```
            other  YoM_H
first      muavia     40
second     muavia     50
third       yazid     61
fourth      walid     95
fifth      hisham    114
sixth      mansur    148
seventh     harun    183
eigth       mamun    203
ninth     Mutasim    220
tenth     Mutasim    254
eleventh    mutaz    260
```

when slice notation `:` is used, then selection happens either by row labels or by integer location

Select rows starting from index of *second* till *tenth*

```
print(df['second':'tenth'])
```

```
             name  age    other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
second      Hasan   47   muavia   MAD        10      3    625     50    670
third      Husain   57    yazid   MAD        11      4    626     61    680
fourth        Ali   57    walid   MAD        34     38    659     95    712
fifth    Muhammad   57   hisham   MAD        19     56    676    114    733
sixth       Jafar   65   mansur   MAD        32     83    702    148    765
seventh      Musa   55    harun   MAD        35    128    745    183    799
eigth        Raza   55    mamun   MAD        20    148    766    203    818
ninth        Taqi   24  Mutasim   MAD        16    195    811    220    835
tenth        Naqi   40  Mutasim   MAD        34    212    828    254    868

         dynasty cityd
second    umayad   MAD
third     umayad   KBL
fourth    umayad   MAD
fifth     umayad   MAD
sixth    abbasid   MAD
seventh  abbasid   BGD
eigth    abbasid   MAS
ninth    abbasid   BGH
tenth    abbasid   SAM
```

Select every second row starting from 3rd till 7th

```
print(df[2:6:2])
```

```
          name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
third    Husain   57   yazid   MAD        11      4    626     61    680
fifth  Muhammad   57  hisham   MAD        19     56    676    114    733

      dynasty cityd
third  umayad   KBL
fifth  umayad   MAD
```

However, there are more specific methods for indexing and slicing a dataframe. These are `loc`, `iloc`, `at` and `iat`. `at` and `iat` are meant to access a scalar, i.e, a single element in the dataframe, while `loc` and `iloc` are used to access several elements at the same time, potentially to perform vectorized operations

loc

- only work on index
- label based

It is used when we want to select rows or columns from a dataframe using the names of columns or the name of index.

The index operator `[]` after `.loc` can have two values/identifiers separated by comma ",". The first identifier (before comma) tells which row/rows we want to select and second identifier tells, which columns we want to select.

For example if we want to select a row whose index is "third", we can use `loc`.

```
print(df.loc['third'])
```

```
name        Husain
age             57
other         yazid
cityb           MAD
duration         11
YoB_H             4
YOB_G           626
YoM_H            61
YoM_G           680
dynasty     umayad
cityd           KBL
Name: third, dtype: object
```

Above we did not specify the second identifier i.e. there is no comma. This is because the if we don't specify the columns, it will give all the columns.

We can select multiple rows with .loc with a list of strings

```
print(df.loc[['second', 'fourth', 'sixth']])
```

```
         name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
second  Hasan   47  muavia   MAD        10      3    625     50    670
fourth   Ali   57   walid   MAD        34     38    659     95    712
sixth  Jafar   65  mansur   MAD        32     83    702    148    765
```

(continues on next page)

(continued from previous page)

```
        dynasty cityd
second   umayad   MAD
fourth   umayad   MAD
sixth   abbasid   MAD
```

Selecting multiple rows with .loc with slice notation `:`

```
print(df.loc['second':'fifth'])
```

```
           name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
second     Hasan   47  muavia   MAD        10      3    625     50    670
third     Husain   57   yazid   MAD        11      4    626     61    680
fourth       Ali   57   walid   MAD        34     38    659     95    712
fifth   Muhammad   57  hisham   MAD        19     56    676    114    733

       dynasty cityd
second  umayad   MAD
third   umayad   KBL
fourth  umayad   MAD
fifth   umayad   MAD
```

In following code, we simultaneously select rows and columns by their labels. Before comman, we tell which rows we want and after comma we tell which columns we want.

```
print(df.loc[['fifth', 'sixth'], 'other':])
```

```
        other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  dynasty cityd
fifth  hisham   MAD        19     56    676    114    733   umayad   MAD
sixth  mansur   MAD        32     83    702    148    765  abbasid   MAD
```

If we want to select all the rows, we can use colon i.e. `:`.

```
print(df.loc[:, 'name':'cityd':2])
```

```
              name    other  duration  YOB_G  YoM_G cityd
first          Ali   muavia        29    600    661   NJF
second       Hasan   muavia        10    625    670   MAD
third       Husain    yazid        11    626    680   KBL
fourth         Ali    walid        34    659    712   MAD
fifth     Muhammad   hisham        19    676    733   MAD
sixth        Jafar   mansur        32    702    765   MAD
seventh       Musa    harun        35    745    799   BGD
eigth         Raza    mamun        20    766    818   MAS
ninth         Taqi  Mutasim        16    811    835   BGH
tenth         Naqi  Mutasim        34    828    868   SAM
eleventh    Askari    mutaz         6    844    874   SAM
```

Above we wanted to select all the rows (indicated by `:`) and every second columns starting from *name* to *cityd*.

We can also select rows/columns with conditions. For example if we want rows where *age* is above 50, we can do as below

```
print(df.loc[df['age']>50])
```

```
             name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
first         Ali   63  muavia   MKH        29    -22    600     40    661
third      Husain   57   yazid   MAD        11      4    626     61    680
fourth        Ali   57   walid   MAD        34     38    659     95    712
fifth    Muhammad   57  hisham   MAD        19     56    676    114    733
sixth       Jafar   65  mansur   MAD        32     83    702    148    765
seventh      Musa   55   harun   MAD        35    128    745    183    799
eigth        Raza   55   mamun   MAD        20    148    766    203    818

         dynasty cityd
first       None   NJF
third     umayad   KBL
fourth    umayad   MAD
fifth     umayad   MAD
sixth    abbasid   MAD
seventh  abbasid   BGD
eigth    abbasid   MAS
```

In above code, `df['age']>50`, is the condition. The output of `df['age']>50` is a boolean array. Thus when we pass a boolean array to `loc`, it returns us rows based upon the specified condition.

We can have multiple conditions as well

```
print(df.loc[(df['age']>50) & (df['duration']>30)])
```

```
          name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
fourth     Ali   57   walid   MAD        34     38    659     95    712
sixth    Jafar   65  mansur   MAD        32     83    702    148    765
seventh   Musa   55   harun   MAD        35    128    745    183    799

         dynasty cityd
fourth    umayad   MAD
sixth    abbasid   MAD
seventh  abbasid   BGD
```

We can not do above conditioning with strings. What if we want all rows where *other* is either muavia or yazid. In such a case we can provide all the values as a list inside the `isin` method.

```
print(df.loc[df['other'].isin(['muavia', 'yazid'])])
```

```
          name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
first      Ali   63  muavia   MKH        29    -22    600     40    661
second   Hasan   47  muavia   MAD        10      3    625     50    670
third   Husain   57   yazid   MAD        11      4    626     61    680

       dynasty cityd
first      None   NJF
second   umayad   MAD
third    umayad   KBL
```

We can even combine boolean indexing/condition with label based indexing.

```
print(df.loc[df['age'] > 30, ['other', 'YoM_H']])
```

```
           other  YoM_H
first     muavia     40
second    muavia     50
third      yazid     61
fourth     walid     95
fifth     hisham    114
sixth     mansur    148
seventh    harun    183
eigth      mamun    203
tenth    Mutasim    254
```

Question: Write the names of the people who were in *umayad* dynasty using loc?

iloc

- integer location based
- work on position

`iloc` is used to select rows and columns from dataframe by their location/index/position value. If we don't know the actual names of columns and just want the columns by their locations/position, we can use iloc. For example if we want the last row from dataframe, we can do as below

```
print(df.iloc[-1])
```

```
name         Askari
age              27
other         mutaz
cityb           MAD
duration          6
YoB_H           232
YOB_G           844
YoM_H           260
YoM_G           874
dynasty     abbasid
cityd           SAM
Name: eleventh, dtype: object
```

If we want the last column, we can do as below

```
print(df.iloc[:, -1])
```

```
first       NJF
second      MAD
third       KBL
fourth      MAD
fifth       MAD
sixth       MAD
seventh     BGD
eigth       MAS
```

(continues on next page)

(continued from previous page)

```
ninth       BGH
tenth       SAM
eleventh    SAM
Name: cityd, dtype: object
```

The : above tells that we want all rows.

If we want 5th row, we can dow as below

```
print(df.iloc[4])
```

```
name        Muhammad
age               57
other          hisham
cityb             MAD
duration           19
YoB_H              56
YOB_G             676
YoM_H             114
YoM_G             733
dynasty       umayad
cityd             MAD
Name: fifth, dtype: object
```

If we want to select multiple rows, we need to pass a list.

Select thrid and second last row

```
print(df.iloc[[2, -2]])
```

```
         name  age    other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
third  Husain   57    yazid   MAD        11      4    626     61    680
tenth    Naqi   40  Mutasim   MAD        34    212    828    254    868

       dynasty cityd
third   umayad   KBL
tenth  abbasid   SAM
```

Selecting multiple rows with .iloc with slice notation

Select every second row starting from first till 8th

```
print(df.iloc[:7:2])
```

```
             name  age   other cityb  duration  YoB_H  YOB_G  YoM_H  YoM_G  \
first         Ali   63  muavia   MKH        29    -22    600     40    661
third      Husain   57   yazid   MAD        11      4    626     61    680
fifth    Muhammad   57  hisham   MAD        19     56    676    114    733
seventh      Musa   55   harun   MAD        35    128    745    183    799

         dynasty cityd
first       None   NJF
third     umayad   KBL
```

(continues on next page)

(continued from previous page)

```
fifth     umayad   MAD
seventh  abbasid   BGD
```

Simultaneous selection of rows and columns

Select second and fifth row and third column

```
print(df.iloc[[1,4], 2])
```

```
second    muavia
fifth     hisham
Name: other, dtype: object
```

As we did with `loc`, we can also use a boolean array for selection to `iloc`.

Select 3rd and fifth column but where age is greater than 30

```
print(df.iloc[(df['age'] > 30).values, [2, 4]])
```

```
           other  duration
first     muavia        29
second    muavia        10
third      yazid        11
fourth     walid        34
fifth     hisham        19
sixth     mansur        32
seventh    harun        35
eigth      mamun        20
tenth    Mutasim        34
```

Question: Select first and last rows and from first and last columns using `iloc`

at

Selection with .at is nearly identical to .loc but it only selects a single 'cell' in your DataFrame. We usually refer to this cell as a scalar value. To use .at, pass it both a row and column label separated by a comma.

```
print(df.at['sixth', 'duration'])
```

```
32
```

iat

Selection with *iat* is nearly identical to *iloc* but it only selects a single scalar value. You must pass it an integer for both the row and column locations

```
print(df.iat[2, 5])
```

```
4
```

Question: Calculate the average age of people in *umayad* and *abbasid* dynasties using `loc` and `iloc`?

Question: Calculate the *years in office* by subtracting *YoM_H* from *YoM_H* of the above row and find out which one had the longest and shortest stay in office? Since there is no row above the first row, consider 11 as the *YoM_H* of the row above.

Question: Who lived the longest and shortest?

Total running time of the script: (0 minutes 0.057 seconds)

5.3 working with time series

Important: This lesson is still under development.

In this file you will learn about following concepts of pandas

- DateTimeIndex
- TimeStamp
- freq
- Timedelta
- offsets
- resampling
- Period

```
import numpy as np
import pandas as pd

pd.set_option('display.max_rows', 20)

print(np.__version__)
print(pd.__version__)
```

```
1.26.4
1.5.3
```

Let's define a dataframe and check its index

```
df = pd.DataFrame(np.arange(31))

print(df)
```

```
     0
0    0
1    1
2    2
3    3
4    4
..  ..
26  26
27  27
```

(continues on next page)

(continued from previous page)

```
28  28
29  29
30  30

[31 rows x 1 columns]
```

Since dataframe is nothing but numpy arrays with indexes which means each row and column has a label (index). Therefore, we can also interpret dataframes as indexed numpy arrays. When we create a dataframe, pandas automatically assigns a suitable index (row labels) to it.

```
print(df.index)
```

```
RangeIndex(start=0, stop=31, step=1)
```

The index is a range from 0 to 1 with step size of 1 and is of of type `RangeIndex`.

we can verify the type of index

```
print(type(df.index))
```

```
<class 'pandas.core.indexes.range.RangeIndex'>
```

DateTimeIndex

Let's define a more useful index for the dataframe i.e., dates with daily time step

```
index = [f"2011-01-{i}" for i in range(1, 32)]

print(index)
```

```
['2011-01-1', '2011-01-2', '2011-01-3', '2011-01-4', '2011-01-5', '2011-01-6', '2011-01-7
↪', '2011-01-8', '2011-01-9', '2011-01-10', '2011-01-11', '2011-01-12', '2011-01-13',
↪'2011-01-14', '2011-01-15', '2011-01-16', '2011-01-17', '2011-01-18', '2011-01-19',
↪'2011-01-20', '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24', '2011-01-25',
↪'2011-01-26', '2011-01-27', '2011-01-28', '2011-01-29', '2011-01-30', '2011-01-31']
```

At this point the *index* is a list of strings where each string indicates a day/date.

Now let's assign this index to our dataframe

```
df.index = index

print(df)
```

```
             0
2011-01-1    0
2011-01-2    1
2011-01-3    2
2011-01-4    3
2011-01-5    4
...         ..
```

(continues on next page)

(continued from previous page)

```
2011-01-27  26
2011-01-28  27
2011-01-29  28
2011-01-30  29
2011-01-31  30

[31 rows x 1 columns]
```

We can see that the index of our the dataframe is now the date. But does pandas recognizes this new index as date or does it considers it still as strings?

```
print(type(df.index))
```

```
<class 'pandas.core.indexes.base.Index'>
```

So it turns out that pandas does not recognize the index as date/time

Therefore, we can explicitly tell pandas that the new index is a date and time index

```
index = pd.to_datetime(index)

print(index)
```

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-15', '2011-01-16',
               '2011-01-17', '2011-01-18', '2011-01-19', '2011-01-20',
               '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28',
               '2011-01-29', '2011-01-30', '2011-01-31'],
              dtype='datetime64[ns]', freq=None)
```

The `to_datetime` function of pandas converts an array of dates into DateTimeIndex object. It can accepts dates in a wide range of formats. We can verify the type of our new index.

```
print(type(index))
```

```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

now we have created an index whose type is DateTimeIndex i.e. pandas recognizes it as date/time. Let's assign this as index to the dataframe.

```
df.index = index

print(df.index)
```

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-15', '2011-01-16',
               '2011-01-17', '2011-01-18', '2011-01-19', '2011-01-20',
```

(continues on next page)

(continued from previous page)

```
               '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28',
               '2011-01-29', '2011-01-30', '2011-01-31'],
              dtype='datetime64[ns]', freq=None)
```

So now the type of index of the dataframe is date/time. Now we can perform slicing based upon time, for example we can ask pandas to return rows which are after 15 January 2011 as below

```
print(df[df.index>pd.Timestamp("20110115")])
```

```
             0
2011-01-16  15
2011-01-17  16
2011-01-18  17
2011-01-19  18
2011-01-20  19
2011-01-21  20
2011-01-22  21
2011-01-23  22
2011-01-24  23
2011-01-25  24
2011-01-26  25
2011-01-27  26
2011-01-28  27
2011-01-29  28
2011-01-30  29
2011-01-31  30
```

Had we done it earlier (before converting our index to pd.DateTimeIndex, we would have got error

creating datetime index

Above we converted a normal index which was of type list into DateTimeIndex using `to_datetime` function. We can directly create DateTimeIndex using `date_range` function.

```
index = pd.date_range(start="20110101", freq="D", periods=31)

print(type(index))
```

```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

```
print(index)
```

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-15', '2011-01-16',
               '2011-01-17', '2011-01-18', '2011-01-19', '2011-01-20',
               '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28',
```

(continues on next page)

(continued from previous page)

```
               '2011-01-29', '2011-01-30', '2011-01-31'],
              dtype='datetime64[ns]', freq='D')
```

We can also define the frequency or time-step of our DateTimeIndex.

```
index = pd.date_range(start="20110101", end="20110131", freq="D")

print(type(index))
```

```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

```
print(index)
```

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-15', '2011-01-16',
               '2011-01-17', '2011-01-18', '2011-01-19', '2011-01-20',
               '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28',
               '2011-01-29', '2011-01-30', '2011-01-31'],
              dtype='datetime64[ns]', freq='D')
```

TimeStamps

The *DateTimeIndex* is indeed an array of *TimeStamps* i.e. each member of DateTimeIndex is a TimeStamp.

```
print(df.index[0])
```

```
2011-01-01 00:00:00
```

```
print(type(df.index[0]))
```

```
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

we can check whether a TimeStamp is in a DateTimeIndex or not

```
print(index[1] in index)
```

```
True
```

we can also compare two TimeStamps

```
print(index[0] > index[1])
```

```
False
```

freq

the index (of dataframe) has a special attribute called `freq` which defines the time-step of the index. It is only available for the index of type `DateTimeIndex`.

```
print(df.index.freq)
```

```
None
```

There can be two reasons for the `freq` to be None. Either the data/DateTimeIndex does not have constant time-steps. In such a case freq (time-step) can not be computed. But sometimes even if the index is of type DateTimeIndex and has constant time-step but it can have None freq. This is what happened above. In both cases we can ask pandas to infer the freq/time-step of the index.

```
print(pd.infer_freq(df.index))
```

```
D
```

Now we can assign the frequency to the DataFrame.index (not DataFrame). This is kind of reminding the DataFrame that this is the time-step of your index.

```
df.index.freq = pd.infer_freq(df.index)

print(df.index.freq)
```

```
<Day>
```

we can see once 'reminded', the pandas now tells us the frequency of its index.

```
print(type(df.index.freq))
```

```
<class 'pandas._libs.tslibs.offsets.Day'>
```

```
print(type(df.index.freqstr))
```

```
<class 'str'>
```

forcing a frequency

```
df = pd.DataFrame(np.arange(31), index=pd.date_range("20110101", periods=31, freq="D"))
```

```
print(df)
```

```
            0
2011-01-01  0
2011-01-02  1
2011-01-03  2
2011-01-04  3
2011-01-05  4
```

(continues on next page)

(continued from previous page)

```
...         ..
2011-01-27  26
2011-01-28  27
2011-01-29  28
2011-01-30  29
2011-01-31  30

[31 rows x 1 columns]
```

```
print(df.index.freq)
```

```
<Day>
```

```
df = df.drop(labels="2011-01-03")

print(df)
```

```
             0
2011-01-01   0
2011-01-02   1
2011-01-04   3
2011-01-05   4
2011-01-06   5
...         ..
2011-01-27  26
2011-01-28  27
2011-01-29  28
2011-01-30  29
2011-01-31  30

[30 rows x 1 columns]
```

```
print(df.index.freq)
```

```
None
```

```
pd.infer_freq(df.index)
```

if we forcefully try to assign a frequency, pandas will throw `ValueError`.

```
# Try by uncommenting following line

# df.index.freq = "D"  # -> ValueError
```

Resampling

Resampling means changing the frequency of time series.

One major advantage of having a frequency i.e *freq* attribute defined is that we can easily change the frequency/time-step of the data (time series).

```
df.asfreq('D')
```

Above when we tried to resample our time series data at daily time step, the time steps where we did not have any value, were assigned NaN values.

upsampling

This refers to changing the time step from larger to smaller such as from daily to hourly

```
df = pd.DataFrame(np.random.randint(0, 5, 5),
                  index=pd.date_range("20110101", periods=5, freq="D"),
                  columns=['a'])
print(df)
```

```
            a
2011-01-01  3
2011-01-02  0
2011-01-03  2
2011-01-04  4
2011-01-05  1
```

```
df.resample("6H")
```

```
<pandas.core.resample.DatetimeIndexResampler object at 0x7f135c1fd3a0>
```

Until now we have told pandas to resample at a particular time step but we have not told which method to use. We can as an example use the *mean* to resample.

```
df.resample("6H").mean()
```

But this did not fill the NaNs in our new data.

```
df.resample("6H").ffill()
```

.mean() returns us a pandas object. We can in fact call `ffill` on it as well.

```
df.resample("6H").mean().ffill()
```

A better way to resample is apply some interpolation method. For example linear interpolation.

```
df.resample("6H").interpolate(method="linear")
```

Sometimes, we may wish to equally distribute a quantity during upsampling For example if we have total amount of rainfall for a day, then linearly interpolating daily rainfall values to hourly will be wrong. In such a case we will wish to distribute daily rainfall to equally to hourly steps

```
df1 = df.resample('6H').mean().ffill()
df1['a'] = df1['a'] / df1.groupby('a')['a'].transform(len)  # len/'size'
print(df1)
```

```
                     a
2011-01-01 00:00:00  0.75
2011-01-01 06:00:00  0.75
2011-01-01 12:00:00  0.75
2011-01-01 18:00:00  0.75
2011-01-02 00:00:00  0.00
2011-01-02 06:00:00  0.00
2011-01-02 12:00:00  0.00
2011-01-02 18:00:00  0.00
2011-01-03 00:00:00  0.50
2011-01-03 06:00:00  0.50
2011-01-03 12:00:00  0.50
2011-01-03 18:00:00  0.50
2011-01-04 00:00:00  1.00
2011-01-04 06:00:00  1.00
2011-01-04 12:00:00  1.00
2011-01-04 18:00:00  1.00
2011-01-05 00:00:00  1.00
```

downsampling

It refers to resampling from low time step to high time step e.g. from hourly to daily

```
df = pd.DataFrame(np.random.randint(0, 5, 24),
                  index=pd.date_range("20110101", periods=24, freq="H"),
                  columns=['a'])
print(df)
```

```
                     a
2011-01-01 00:00:00  3
2011-01-01 01:00:00  3
2011-01-01 02:00:00  4
2011-01-01 03:00:00  3
2011-01-01 04:00:00  4
...                 ..
2011-01-01 19:00:00  3
2011-01-01 20:00:00  1
2011-01-01 21:00:00  1
2011-01-01 22:00:00  4
2011-01-01 23:00:00  0

[24 rows x 1 columns]
```

```
df.resample("6H").mean()
```

```
df.resample("6H").sum()
```

inconsistent time step

Sometimes we have quantities, which are not measured at exactly the same frequency where we want. For example below data is measured with inconsistent time steps.

```
df = pd.DataFrame([np.nan, 1100, 1400, np.nan, 14000],
                  index=pd.to_datetime(["2011-05-25 10:00:00",
                                        "2011-05-25 16:40:00",
                                        "2011-05-25 17:06:00",
                                        "2011-05-25 17:10:00",
                                        "2011-05-25 17:24:00"]),
                  columns=['a'])

print(df)
```

```
                            a
2011-05-25 10:00:00       NaN
2011-05-25 16:40:00    1100.0
2011-05-25 17:06:00    1400.0
2011-05-25 17:10:00       NaN
2011-05-25 17:24:00   14000.0
```

Our target is to convert this data to 6 minute. A naive way would be to change the frequency and do not fill the new nans.

```
df.resample('6Min').first()
```

You see the number of values change from 5 to 75

a better option will be to do backfill or forward fill

```
df.resample('6min').bfill(limit=1)
```

it will be even better to do a linear interpolation between available values.

```
df.resample('6min').interpolate()
```

```
df.resample('6min').interpolate('nearest')
```

Period

A Period is an interval between two TimeStamps. Therefore a Period has `start_time` and `end_time` attributes

```
p = pd.Period("1979-02-01")

print(type(p))
```

```
<class 'pandas._libs.tslibs.period.Period'>
```

```
print(p.start_time)
```

```
1979-02-01 00:00:00
```

```
print(p.end_time)
```

```
1979-02-01 23:59:59.999999999
```

```
print(type(p.start_time)), print(type(p.end_time))
```

```
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
<class 'pandas._libs.tslibs.timestamps.Timestamp'>

(None, None)
```

```
print(p.freq)
```

```
<Day>
```

PeriodIndex

Similar to the concept of DateTimeIndex is the concept of PeriodIndex. Just as a DateTimeIndex can be considered as an array of TimeStamps, a PeriodIndex is array of Period.

```
pidx = pd.period_range("20110101", "20121231", freq="M")
print(pidx)
```

```
PeriodIndex(['2011-01', '2011-02', '2011-03', '2011-04', '2011-05', '2011-06',
             '2011-07', '2011-08', '2011-09', '2011-10', '2011-11', '2011-12',
             '2012-01', '2012-02', '2012-03', '2012-04', '2012-05', '2012-06',
             '2012-07', '2012-08', '2012-09', '2012-10', '2012-11', '2012-12'],
            dtype='period[M]')
```

```
print(type(pidx))
```

```
<class 'pandas.core.indexes.period.PeriodIndex'>
```

each member of PeriodIndex array i.e., **pidx** is a Period

```
print(type(pidx[0]))
```

```
<class 'pandas._libs.tslibs.period.Period'>
```

For an overview of difference between TimeStamp and PeriodIndex, see this

Total running time of the script: (0 minutes 0.215 seconds)

5.4 reading/writing

This file describes how to read data from files and write data into files using pandas.

Important: This lesson is still under development.

```
import pandas as pd
```

```
df = pd.read_csv("https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/
↪datasets/arg_busan.csv")

type(df)
```

```
df
```

```
df.to_csv("arg_busan.csv")
```

The index of df was 0,1,2,... By default, to_csv function writes the index to csv

```
print(df.index)
```

```
RangeIndex(start=0, stop=1446, step=1)
```

```
print(df.index.name)
```

```
None
```

we can avoid writing the index to csv file by setting `index=False`.

```
df.to_csv("arg_busan.csv", index=False)
```

we can also explicitly tell pandas what label for index to use when writing the index to csv file.

```
df.to_csv("arg_busan.csv", index_label="index")
```

if we want to save a dataframe to Excel file we can do it as following

```
df.to_excel("arg_busan.xlsx")  # we must have ``openpyxl`` package for that
```

we can define the sheet name and exlcude the index as following

```
df.to_excel("arg_busan.xlsx", index=False, sheet_name="data")
```

to read the excel file as dataframe we can make use of read_excel function

```
df = pd.read_excel("arg_busan.xlsx")
print(df)
```

```
               index    tide_cm  wat_temp_c    sal_psu  air_temp_c  ...  \
0     6/19/2018 0:00  36.407149   19.321232  33.956058   19.780000  ...
1     6/19/2018 0:30  35.562515   19.320124  33.950508   19.093333  ...
```

(continues on next page)

(continued from previous page)

```
2     6/19/2018 1:00  34.808016   19.319666  33.942532   18.733333  ...
3     6/19/2018 1:30  30.645216   19.320406  33.931263   18.760000  ...
4     6/19/2018 2:00  26.608980   19.326729  33.917961   18.633333  ...
...               ...        ...        ...        ...         ...  ...
1441  9/7/2019 22:00  -3.989912   20.990612  33.776449   23.700000  ...
1442  9/7/2019 22:30  -2.807042   21.012014  33.702310   23.620000  ...
1443  9/7/2019 23:00  -3.471326   20.831739  33.726177   23.666667  ...
1444  9/7/2019 23:30   0.707771   21.006086  33.716274   23.633333  ...
1445   9/8/2019 0:00   1.011731   20.896149  33.729773   23.600000  ...

      aac_coppml  Total_otus  otu_5575  otu_273  otu_94
0            NaN         NaN       NaN      NaN     NaN
1            NaN         NaN       NaN      NaN     NaN
2            NaN         NaN       NaN      NaN     NaN
3            NaN         NaN       NaN      NaN     NaN
4            NaN         NaN       NaN      NaN     NaN
...          ...         ...       ...      ...     ...
1441         NaN         NaN       NaN      NaN     NaN
1442         NaN         NaN       NaN      NaN     NaN
1443         NaN         NaN       NaN      NaN     NaN
1444         NaN         NaN       NaN      NaN     NaN
1445         NaN         NaN       NaN      NaN     NaN

[1446 rows x 26 columns]
```

we can tell which column should be used as index for the dataframe

```
df = pd.read_excel("arg_busan.xlsx", index_col="index")
print(df)
```

```
                  tide_cm  wat_temp_c    sal_psu  air_temp_c  pcp_mm  ...  \
index                                                                 ...
6/19/2018 0:00  36.407149   19.321232  33.956058   19.780000     0.0  ...
6/19/2018 0:30  35.562515   19.320124  33.950508   19.093333     0.0  ...
6/19/2018 1:00  34.808016   19.319666  33.942532   18.733333     0.0  ...
6/19/2018 1:30  30.645216   19.320406  33.931263   18.760000     0.0  ...
6/19/2018 2:00  26.608980   19.326729  33.917961   18.633333     0.0  ...
...                   ...         ...        ...         ...     ...  ...
9/7/2019 22:00  -3.989912   20.990612  33.776449   23.700000     0.0  ...
9/7/2019 22:30  -2.807042   21.012014  33.702310   23.620000     0.0  ...
9/7/2019 23:00  -3.471326   20.831739  33.726177   23.666667     0.0  ...
9/7/2019 23:30   0.707771   21.006086  33.716274   23.633333     0.0  ...
9/8/2019 0:00    1.011731   20.896149  33.729773   23.600000     0.0  ...

                aac_coppml  Total_otus  otu_5575  otu_273  otu_94
index
6/19/2018 0:00         NaN         NaN       NaN      NaN     NaN
6/19/2018 0:30         NaN         NaN       NaN      NaN     NaN
6/19/2018 1:00         NaN         NaN       NaN      NaN     NaN
6/19/2018 1:30         NaN         NaN       NaN      NaN     NaN
6/19/2018 2:00         NaN         NaN       NaN      NaN     NaN
...                    ...         ...       ...      ...     ...
```

(continues on next page)

(continued from previous page)

```
9/7/2019 22:00         NaN         NaN       NaN      NaN     NaN
9/7/2019 22:30         NaN         NaN       NaN      NaN     NaN
9/7/2019 23:00         NaN         NaN       NaN      NaN     NaN
9/7/2019 23:30         NaN         NaN       NaN      NaN     NaN
9/8/2019 0:00          NaN         NaN       NaN      NaN     NaN

[1446 rows x 25 columns]
```

```
print(type(df.index))
```

```
<class 'pandas.core.indexes.base.Index'>
```

Although we index of dataframe is date and time but pandas does not recognize it as data and time but it recognizes it just as numbers

```
df = pd.read_excel("arg_busan.xlsx", index_col="index", parse_dates=True)

print(df)
```

```
                     tide_cm  wat_temp_c    sal_psu  air_temp_c  pcp_mm  \
index
2018-06-19 00:00:00  36.407149   19.321232  33.956058   19.780000     0.0
2018-06-19 00:30:00  35.562515   19.320124  33.950508   19.093333     0.0
2018-06-19 01:00:00  34.808016   19.319666  33.942532   18.733333     0.0
2018-06-19 01:30:00  30.645216   19.320406  33.931263   18.760000     0.0
2018-06-19 02:00:00  26.608980   19.326729  33.917961   18.633333     0.0
...                        ...         ...        ...         ...     ...
2019-09-07 22:00:00  -3.989912   20.990612  33.776449   23.700000     0.0
2019-09-07 22:30:00  -2.807042   21.012014  33.702310   23.620000     0.0
2019-09-07 23:00:00  -3.471326   20.831739  33.726177   23.666667     0.0
2019-09-07 23:30:00   0.707771   21.006086  33.716274   23.633333     0.0
2019-09-08 00:00:00   1.011731   20.896149  33.729773   23.600000     0.0

                     ...  aac_coppml  Total_otus  otu_5575  otu_273  otu_94
index                ...
2018-06-19 00:00:00  ...         NaN         NaN       NaN      NaN     NaN
2018-06-19 00:30:00  ...         NaN         NaN       NaN      NaN     NaN
2018-06-19 01:00:00  ...         NaN         NaN       NaN      NaN     NaN
2018-06-19 01:30:00  ...         NaN         NaN       NaN      NaN     NaN
2018-06-19 02:00:00  ...         NaN         NaN       NaN      NaN     NaN
...                  ...         ...         ...       ...      ...     ...
2019-09-07 22:00:00  ...         NaN         NaN       NaN      NaN     NaN
2019-09-07 22:30:00  ...         NaN         NaN       NaN      NaN     NaN
2019-09-07 23:00:00  ...         NaN         NaN       NaN      NaN     NaN
2019-09-07 23:30:00  ...         NaN         NaN       NaN      NaN     NaN
2019-09-08 00:00:00  ...         NaN         NaN       NaN      NaN     NaN

[1446 rows x 25 columns]
```

Now the index of dataframe is read as `DateTimeIndex`

```
print(type(df.index))

#%%
```

```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Total running time of the script: (0 minutes 3.405 seconds)

5.5 apply

Important: This lesson is still under development.

```
from datetime import timedelta
import pandas as pd
import numpy as np


def filterByTime(
        pdf: pd.Series,
        start: pd.Timestamp,
        end: pd.Timestamp
) -> pd.Series:
    """
    Filter subset by datetime specified its header

    Parameters:
        pdf :
            source data to filter
        start :
            start date / time
        end :
        end  date/time

    return:
      pandas.Series object
    """
    return pdf.loc[start:end - timedelta(seconds=1)]


def interpolateByTime(
        pdf0: pd.Series,
        pdf1: pd.Series,
        freq: str = "H"
) -> float:
    """
    Interpolate data at 00:00

    Parameters:
      pdf0: data befor 0:00
        Type: pandas.DataFrame object
```

(continues on next page)

(continued from previous page)

```
      pdf1: data after 0:00
        Type: pandas.DataFrame object
      freq
    return:
      value with a type of float
    """
    replace = dict(minute=0, second=0)
    if freq == "D":
        replace = dict(hour=0, minute=0)
    v0, t0 = pdf0.iloc[-1], pdf0.index[-1]
    v1, t1 = pdf1.iloc[0], pdf1.index[0]

    t = t1.replace(**replace)

    dt0 = (t - t0).total_seconds() / 3600
    dt1 = (t1 - t).total_seconds() / 3600
    v = (v0 * dt1 + v1 * dt0) / (dt0 + dt1)
    return np.float32(v)


def prepend(historical_data, subdata, freq: str = "H"):
    """
    Add first row at 0:00. When length of pdf0 greater than 1, interpolate it. Or, use␣
↪first row of pdf1.

    Parameters:
        historical_data: data befor 0:00
            Type: pandas.DataFrame object
        subdata: data after 0:00
            Type: pandas.DataFrame object
        freq

    return:
      pandas.DataFrame object
    """
    first = subdata.iloc[:1].copy()
    if freq == "H":
        # e.g 2024-01-01 04:10 -> 2023-01-01 04:00
        first.index = first.index.where([0], first.index[0].replace(minute=0, second=0))
    else:
        # e.g. 2024-01-02 08:00:00 -> 2024-01-02 00:00:00
        first.index = first.index.where([0], first.index[0].replace(hour=0, minute=0,␣
↪second=0))
    if len(historical_data) > 1:
        first.loc[first.index[0]] = interpolateByTime(historical_data, subdata,␣
↪freq=freq)
    pdf = pd.concat([subdata, first]).sort_index()
    return pdf


def append(
        subdata: pd.Series,
```

(continues on next page)

(continued from previous page)

```
        future_data: pd.Series,
        freq: str = "H"):
    """
    Add last row at 0:00. When length of pdf1 greater than 1, interpolate it.
    Or, use last row of pdf0.

    Parameters:
        subdata: data befor for the current step (day, hour)
            Type: pandas.DataFrame object
        future_data: data after 0:00
            Type: pandas.DataFrame object
        freq :
    return:
      pandas.DataFrame object
    """
    last = subdata.iloc[-1:].copy()
    if freq == "H":
        # e.g 2024-01-01 02:45 -> 2023-01-01 03:00
        last.index = last.index.where([0],
                                      last.index[0].replace(
                                          minute=0, second=0) + timedelta(hours=1))
    else:
        last.index = last.index.where([0],
                                      last.index[0].replace(
                                          hour=0, minute=0, second=0) +␣
↪timedelta(days=1))
    if len(future_data) > 1:
        last.loc[last.index[0]] = interpolateByTime(subdata, future_data, freq=freq)
    pdf = pd.concat([subdata, last]).sort_index()
    return pdf


def weightsCalculator(index, freq: str = "H"):
    """
    Calculate weights of time for variable(water level or flow). Actuall, they
    have a unit of hour.

    Parameters:
        index: time corresponding to the variable data
          Type: list of datetime or numpy array of datetime
        freq :
    return:
      numpy array in float/int with a unit of hour
    """
    _format = 'm'
    total = 120
    if freq == "D":
        _format = 'h'
        total = 48

    dt = np.array(index[1:]) - np.array(index[:-1])
    dt = [i / np.timedelta64(1, _format) for i in dt]  # convert timedelta object to␣
↪number of hours
```

(continues on next page)

(continued from previous page)

```
    weights = np.array([0] + dt) + np.array(dt + [0])
    assert int(round(weights.sum())) == total, int(round(weights.sum()))
    return weights


def tw_resampler(
        subdata: pd.Series,
        whole_data: pd.Series,
        freq: str = "H",
) -> float:
    """
    resampler for time weighted average. Resamplers from sub-daily to daily or
    sub-hourly to hourly considering time weighted average instead of simple average.

    Parameters
    ----------
    subdata : pd.Series
        The data for the current time step (day/hour). The index of subdata must be
        pandas DatetimeIndex.
    whole_data : pd.Series
        is used to get data for historical and next step for interpolation at start and␣
↪end
        of current step if the data at start and end of current step is not available.
    freq : str
        must be either ``H`` or ``D``

    index = pd.to_datetime([
    '2024-01-01 00:00:00', '2024-01-01 08:00:00',
    '2024-01-01 02:00:00', '2024-01-01 08:20:00',
    '2024-01-01 02:45:00', '2024-01-01 09:45:00',
    '2024-01-01 04:10:00', '2024-01-01 09:50:00',
    '2024-01-01 04:20:00', '2024-01-01 11:10:00',
    '2024-01-01 04:35:00', '2024-01-01 11:15:00',
    '2024-01-01 04:45:00', '2024-01-01 11:45:00',
    '2024-01-01 04:55:00', '2024-01-01 12:15:00',
    '2024-01-01 05:15:00', '2024-01-01 12:25:00',
    '2024-01-01 06:15:00', '2024-01-01 12:35:00',

    '2024-01-01 13:00:00', '2024-01-01 19:00:00',
    '2024-01-01 13:10:00', '2024-01-01 19:10:00',
    '2024-01-01 13:45:00', '2024-01-01 19:45:00',
    '2024-01-01 14:30:00', '2024-01-01 19:50:00',
    '2024-01-01 14:50:00', '2024-01-01 21:30:00',
    '2024-01-01 15:15:00', '2024-01-01 21:45:00',
    '2024-01-01 15:35:00', '2024-01-01 22:15:00',
    '2024-01-01 17:15:00', '2024-01-01 22:55:00',
    '2024-01-01 17:16:00', '2024-01-01 23:17:00',
    '2024-01-01 17:17:00', '2024-01-01 23:19:00',
        ])

    df = pd.DataFrame(
    np.arange(len(index)),
```

(continues on next page)

(continued from previous page)

```
    index=index,
    columns=['value']
    ).astype(np.float32)
    val = df['value'].resample('H').apply(lambda subdata: tw_resampler_hourly(subdata, df[
↪'value'].sort_index()))

    np.testing.assert_array_almost_equal([val.sum()], [327.425], decimal=4)
    # resampling from sub-daily to daily time-step

    index = pd.to_datetime([
    '2024-01-01 00:00:00', '2024-01-02 08:00:00',
    '2024-01-01 02:00:00', '2024-01-02 08:20:00',
    '2024-01-01 02:45:00', '2024-01-02 09:45:00',
    '2024-01-01 04:10:00', '2024-01-02 09:50:00',
    '2024-01-01 04:20:00', '2024-01-02 11:10:00',
    '2024-01-01 04:35:00', '2024-01-02 11:15:00',
    '2024-01-01 04:45:00', '2024-01-02 11:45:00',
    '2024-01-01 04:55:00', '2024-01-02 12:15:00',
    '2024-01-01 05:15:00', '2024-01-02 12:25:00',
    '2024-01-01 06:15:00', '2024-01-02 12:35:00',

    '2024-01-03 13:00:00', '2024-01-04 19:00:00',
    '2024-01-03 13:10:00', '2024-01-04 19:10:00',
    '2024-01-03 13:45:00', '2024-01-04 19:45:00',
    '2024-01-03 14:30:00', '2024-01-05 19:50:00',
    '2024-01-03 14:50:00', '2024-01-07 21:30:00',
    '2024-01-03 15:15:00', '2024-01-08 21:45:00',
    '2024-01-03 15:35:00', '2024-01-08 22:15:00',
    '2024-01-03 17:15:00', '2024-01-08 22:55:00',
    '2024-01-03 17:16:00', '2024-01-08 23:17:00',
    '2024-01-03 17:17:00', '2024-01-08 23:19:00',
        ])

    df = pd.DataFrame(
    np.arange(len(index)),
    index=index,
    columns=['value']
    ).astype(np.float32)
    df['value'].resample('D').apply(lambda subdata: tw_resampler(subdata, df['value'].sort_
↪index(), 'D'))

    """

    REPLACE = {
        'D': {'hour': 0, 'minute': 0, 'second': 0},
        'H': {'minute': 0, 'second': 0}
    }

    DELTA = {
        "D": {"d0": dict(days=1), "d3": dict(days=2)},
        "H": {"d0": dict(hours=1), "d3": dict(hours=2)}
    }
```

(continues on next page)

(continued from previous page)

```
    # case 0: do nothing
    if len(subdata) == 0:
        return np.nan

    # case 1: use it as the daily average if only one gauged data
    if len(subdata) == 1:
        record = round(subdata.iloc[0], 4)
    else:
        d1 = subdata.index[0].replace(**REPLACE[freq])  # start of the current step (d1)
        d0 = d1 - timedelta(**DELTA[freq]['d0'])  # start of the previous step (d0)
        d2 = d1 + timedelta(**DELTA[freq]['d0'])  # start of the next step (d2)
        d3 = d1 + timedelta(**DELTA[freq]['d3'])  # start of the next two steps (d3)
        # first index is not starts with 00:00
        if subdata.index[0] != d1:
            historical_data = whole_data.loc[d0:d1 - timedelta(seconds=1)].dropna()  # .
↪sort_index()
            subdata = prepend(historical_data, subdata, freq=freq)

        # last index is not 00:00
        if subdata.index[-1] != d2:
            future_data = whole_data.loc[d2:d3 - timedelta(seconds=1)].dropna()  # .sort_
↪index()
            subdata = append(subdata, future_data, freq=freq)

        # weights
        weights = weightsCalculator(subdata.index.to_numpy(), freq=freq)

        # daily average
        record = np.average(subdata.values, weights=weights)

    return record




# print("using pallalel")

# start = time.time()
# resampler = site_data['00060'].iloc[0:10_000].resample('H')
# wh_data = site_data['00060'].iloc[0:10_000].sort_index()
# with ThreadPoolExecutor(32) as executor:
#     results = list(executor.map(tw_resampler,
#                                 [group for _, group in resampler],
#                                 [wh_data]*len(resampler)))
# val = pd.Series(results, index=resampler.groups.keys())
# print(time.time() - start)


# wh_data = df['value'].sort_index()
# results = []
# resampler = df['value'].resample('H')
```

(continues on next page)

(continued from previous page)

```
# for _, group in resampler:
#    results.append(tw_resampler(group, wh_data))
# val = pd.Series(results, index=resampler.groups.keys())


# with ProcessPoolExecutor(4) as executor:
#     results = list(executor.map(tw_resampler_hourly,
#                                 [group for _, group in resampler],
#                                 [wh_data]*len(resampler)))
# val = pd.Series(results, index=resampler.groups.keys())
```

Total running time of the script: (0 minutes 0.004 seconds)

5.6 groupby

This lesson shows the usage of groupby in pandas

Important: This lesson is still under development.

The material in this lesson is mostly inspired from pandas documentation for groupby and realpython .

```
import time

import numpy as np
import pandas as pd

print(time.asctime())
print(np.__version__)
print(pd.__version__)
```

```
Mon Nov 11 19:32:28 2024
1.26.4
1.5.3
```

```
url = "https://danepubliczne.imgw.pl/data/dane_pomiarowo_obserwacyjne/dane_hydrologiczne/
↪dobowe/2020/codz_2020_01.zip"

df = pd.read_csv(
    url, compression='zip', encoding="ISO-8859-1",  engine='python',
    on_bad_lines="skip",
    names=['stn_id', 'year', 'day', 'water_level_cm', 'q_cms', 'temp_C', 'month'],
    usecols=[0, 3, 5, 6, 7, 8, 9],
    dtype={'stn_id': int, 'year': 'int', 'day': 'int', 'water_level_cm': np.float32, 'q_
↪cms': np.float32, 'temp_C': np.float32, 'month': 'int'},
    parse_dates={'date': ['year', 'month', 'day']},
    index_col='date'
    )

df
```

```
print(df.shape)
```

```
(24913, 4)
```

```
df['stn_id'].unique()
```

```
array([149180020, 149180300, 149180010, 150180060, 150180030, 150170240,
       150170290, 150170130, 150170090, 150170040, 151170030, 151160170,
       151160150, 151160130, 151160060, 151150150, 152150130, 152150050,
       152140130, 152140090, 152140050, 152140060, 152140020, 152140010,
       153140020, 153140030, 153140050, 153140190, 150170160, 150170170,
       149180130, 149180060, 149180070, 149180030, 149180050, 149180040,
       150180040, 150180090, 150180280, 150180130, 150180110, 150180140,
       150180120, 150180080, 150180250, 150180220, 150180180, 150180150,
       150180070, 150180170, 150180160, 150180010, 150170180, 150170110,
       150170080, 150170220, 150180190, 150180100, 150180050, 150180020,
       150180230, 150160190, 150160170, 150160180, 150160220, 150170060,
       150170100, 150170140, 150160210, 150160150, 150160230, 150160200,
       150160110, 150160080, 150160100, 150160270, 150170070, 150170050,
       150170120, 150170150, 150170200, 150170210, 150170010, 150170030,
       150160250, 150160280, 151160230, 150160060, 150160070, 150160120,
       150160160, 151160190, 150160140, 150160130, 150160290, 150160030,
       150160090, 151160180, 150160020, 151170090, 151170050, 151170010,
       151150170, 151160020, 151160050, 151160100, 151160090, 151160070,
       151160040, 151160080, 151150160, 151150180, 151170070, 151170040,
       151160140, 151170080, 151170060, 151160220, 151160200, 151160160,
       151160030, 151160260, 151150190, 152150170, 151160250, 151160010,
       152150060, 150150120, 150150130, 150160010, 150150100, 150150080,
       150150060, 151150140, 151150120, 151150080, 151150050, 151150040,
       152150020, 150150090, 150150110, 150150030, 150150050, 150150070,
       150150190, 150150200, 150150040, 151150130, 150150010, 151150110,
       151150060, 151150100, 151150090, 150150020, 151150070, 151150030,
       150140010, 150140020, 151140060, 151140040, 151140010, 150140100,
       150140030, 151150020, 151140050, 151150010, 151140030, 151140020,
       152140110, 152140080, 150190240, 150190200, 150190150, 150190220,
       151190060, 151180130, 151180100, 151180120, 151180080, 151180110,
       152180120, 152180050, 152170130, 152170080, 152170060, 152170010,
       152160140, 152160100, 152160050, 152150200, 152150110, 152150080,
       152150040, 152150010, 152140070, 150180210, 151190010, 151180090,
       151190020, 151180170, 151180140, 151190030, 151180180, 151180150,
       151190040, 151180160, 152180150, 152180140, 152180110, 152180160,
       152180060, 152170150, 152170140, 152170160, 152170190, 152170120,
       151180070, 151180040, 151180020, 151170110, 151180030, 151180060,
       151180010, 151180050, 152160090, 152160130, 152160060, 152169998,
       152160150, 152170020, 152170050, 152170030, 152170040, 152160110,
       152160120, 152160170, 152160080, 152160040, 152150270, 152150220,
       152150100, 152160030, 152150120, 152180100, 152180090, 152180170,
       152180080, 152180030, 153170100, 153170010, 153160170, 153160160,
       152160070, 152160010, 152150190, 152150140, 152150090, 152180010,
       152180020, 152170100, 152170090, 152170110, 153170040, 153160300,
       153160200, 153160210, 153160180, 153160150, 153160220, 153160260,
       153160250, 153160050, 153160290, 153160270, 153160070, 153160190,
```

(continues on next page)

(continued from previous page)

```
    153160110, 153160060, 153170160, 153170020, 153170150, 153160280,
    153160030, 153160040, 153150120, 153150140, 153150100, 152150240,
    153160010, 153150180, 153150160, 153150150, 153150170, 152150150,
    152150180, 152150230, 152140120, 152140100, 152140030, 152140270,
    153140110, 153140200, 153140100, 153150010, 153140090, 153150060,
    153150020, 153140080, 153140060, 153150090, 153150080, 153150050,
    154150010, 153150030, 153150070, 154150020, 153160080, 153160020,
    154150050, 154150040, 154160080, 154160020, 153160240, 154160010,
    154160030, 154160040, 154160050, 154170040, 154160120, 154160070,
    154160130, 154160100, 154160090, 154160060, 154170120, 154170070,
    154170010, 154160140, 154170030, 154170020, 154170140, 154170230,
    154170150, 154170080, 154170060, 154170050, 154180020, 154170160,
    154170110, 154170090, 154170210, 154170130, 154180050, 154180030,
    154180080, 154180070, 150160040, 149180200, 149180180, 149180160,
    149180140, 149180110, 149180100, 149180080, 149180210, 149180240,
    149190060, 150190140, 150190170, 150190360, 150190260, 149190230,
    150190340, 150200060, 150200100, 150200130, 150200150, 150210020,
    150210150, 150210170, 150210190, 150210180, 151210190, 151210120,
    151210050, 152210040, 152210170, 152200110, 152200150, 152200030,
    152190120, 152190030, 153180090, 153180020, 153180080, 153180100,
    154180150, 154180220, 154180190, 154180200, 154180210, 154190040,
    154190020, 154180160, 154180170, 154180180, 149180120, 149180250,
    149180230, 149190030, 149190010, 149180090, 149180220, 150190060,
    150190050, 150190190, 150190120, 150190130, 150190080, 150190180,
    150190210, 150190010, 150180270, 150180260, 150190070, 150190270,
    150190250, 150190100, 149190050, 149190080, 149190100, 149190120,
    150190160, 149190040, 149190020, 149190090, 149190150, 149190070,
    149190140, 149190290, 149190260, 149190210, 149190180, 149190170,
    149190190, 149190220, 149190200, 149190160, 149190270, 150190310,
    150190330, 149190340, 149200060, 149200040, 149190310, 149200090,
    149200170, 149200080, 149190370, 149190350, 149200130, 150200070,
    150200140, 150200110, 149190280, 149200030, 149200050, 149200140,
    149200160, 149200190, 149200240, 149200230, 149200280, 150200170,
    149190300, 149190360, 149190390, 149190380, 149200020, 149190320,
    149200010, 149200100, 149200110, 149200070, 149200120, 149200180,
    149200150, 149200300, 149200290, 149200220, 149200270, 149200250,
    149200260, 149200200, 149200310, 149200330, 149200320, 150200010,
    150200030, 150200080, 150200020, 150200160, 150200120, 150200040,
    150200090, 150200050, 150210070, 150210010, 150210060, 150210100,
    150210030, 150210050, 149210090, 149210070, 149210050, 149210040,
    150210130, 150210120, 149210010, 149210030, 149210060, 149210020,
    149210100, 149210080, 150210110, 150210140, 150210160, 150210200,
    149220150, 149220130, 149220060, 149220030, 149220040, 149220190,
    150220100, 150220090, 150220070, 150220030, 150210210, 149220180,
    149220140, 149220080, 149220100, 149220110, 149220070, 149220020,
    149220050, 149220160, 149220200, 149220210, 150220140, 150220130,
    149210150, 149210160, 149210110, 149210130, 150220010, 150220080,
    149220010, 149210140, 149210120, 150220060, 150220040, 150220160,
    150220050, 150220110, 150220020, 151210130, 151200100, 151210010,
    151210020, 151210040, 150210090, 151210090, 150210040, 150210080,
    150210220, 151210080, 151220040, 150220170, 150230080, 150230010,
    150230020, 150230040, 151230010, 151220090, 151220010, 150220120,
```

(continues on next page)

(continued from previous page)

```
       150230030, 150230050, 150230060, 151220130, 151220140, 151220070,
       151220100, 151220110, 151220080, 151220120, 151220060, 151220050,
       151200110, 151210060, 151210030, 151210110, 151210100, 151210070,
       150190280, 151190090, 151190100, 151200020, 151200090, 151200120,
       150190350, 151200060, 151190120, 151200070, 151190080, 151190110,
       151200040, 151200080, 152210070, 152210020, 152230120, 152230110,
       152230090, 152230040, 152220080, 153220270, 153220130, 153220070,
       153220010, 153210210, 153210090, 152210060, 152200130, 152230190,
       152230100, 152230060, 152230030, 153230160, 153230170, 153230110,
       153230010, 153230140, 153230080, 153230060, 153220230, 153220180,
       153230070, 153220260, 153220170, 153220100, 153230130, 154220070,
       153220240, 153220300, 153220280, 153220290, 153230050, 153230040,
       153230020, 154220060, 153220110, 153220120, 153220150, 153220190,
       153220320, 153220200, 153220220, 154220020, 154220030, 153220060,
       153220050, 153220080, 153220140, 153220160, 153220310, 153220090,
       154210090, 153210110, 153210130, 153210200, 153210190, 153210170,
       153210220, 154210040, 153210100, 153210050, 153210080, 153210240,
       153210230, 153210180, 153210140, 153210040, 153210120, 153210010,
       153210070, 153200160, 152210100, 153210020, 152210030, 150240020,
       150240010, 151230060, 151230040, 152230080, 152230200, 152220050,
       152229999, 152210090, 152210150, 150230070, 151230050, 151230030,
       151230080, 152230020, 152230070, 152230050, 151230020, 152220060,
       152230010, 152220070, 152220030, 152220090, 152210130, 152220010,
       152210120, 152210140, 152210160, 152210050, 153190160, 152200020,
       152200120, 153200050, 153200020, 152200100, 152200060, 152200080,
       152190050, 152199992, 152200050, 151190070, 152190100, 152200010,
       152200090, 152200070, 152190060, 152190080, 152190110, 152190020,
       152190040, 152180130, 153200010, 153190170, 153190140, 153190120,
       153190090, 153190050, 153180140, 153190080, 153190100, 153190150,
       153190130, 153190180, 153190070, 153190190, 153190060, 153170050,
       153170080, 153170060, 153170120, 153170140, 153179996, 153170070,
       153170110, 153170090, 154170180, 153170130, 153180010, 153180050,
       153180060, 153180040, 153190020, 153180150, 153190010, 154180010,
       153180030, 154180110, 153180110, 154179991, 154170190, 154180230,
       154170240, 154180040, 154180060, 154180270, 154180260, 154190010,
       153190030, 153180130, 153190040, 154190080, 154190060, 154190100,
       154190110, 153200040, 153200030, 154190170, 154190150, 154190160,
       154190140, 154200020, 154200010, 153200070, 154200030, 154210010,
       153200140, 153200090, 154200040, 154210020, 154210030, 154210060,
       154210070, 154210080, 154210100, 154220050, 154220110, 154220010,
       154220100, 154230010, 154220080, 154220090, 154230020, 154230030,
       153230120, 153230090, 154230040, 149220170, 149190240, 149190250])
```

```
df['stn_id'].nunique()
```

```
840
```

```
grouper = df.groupby('stn_id')

grouper
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f1355ab2df0>
```

```
type(grouper)
```

```
grouper.ngroups
```

```
840
```

```
grouper.groups
```

```
{149180010: [2020-11-01 00:00:00, 2020-11-02 00:00:00, 2020-11-03 00:00:00, 2020-11-04␣
↪00:00:00, 2020-11-05 00:00:00, 2020-11-06 00:00:00, 2020-11-07 00:00:00, 2020-11-08␣
↪00:00:00, 2020-11-09 00:00:00, 2020-11-10 00:00:00, 2020-11-11 00:00:00, 2020-11-12␣
↪00:00:00, 2020-11-13 00:00:00, 2020-11-14 00:00:00, 2020-11-15 00:00:00, 2020-11-16␣
↪00:00:00, 2020-11-17 00:00:00, 2020-11-18 00:00:00, 2020-11-19 00:00:00, 2020-11-20␣
↪00:00:00, 2020-11-21 00:00:00, 2020-11-22 00:00:00, 2020-11-23 00:00:00, 2020-11-24␣
↪00:00:00, 2020-11-25 00:00:00, 2020-11-26 00:00:00, 2020-11-27 00:00:00, 2020-11-28␣
↪00:00:00, 2020-11-29 00:00:00, 2020-11-30 00:00:00], 149180020: [2020-11-01 00:00:00,␣
↪2020-11-02 00:00:00, 2020-11-03 00:00:00, 2020-11-04 00:00:00, 2020-11-05 00:00:00,␣
↪2020-11-06 00:00:00, 2020-11-07 00:00:00, 2020-11-08 00:00:00, 2020-11-09 00:00:00,␣
↪2020-11-10 00:00:00, 2020-11-11 00:00:00, 2020-11-12 00:00:00, 2020-11-13 00:00:00,␣
↪2020-11-14 00:00:00, 2020-11-15 00:00:00, 2020-11-16 00:00:00, 2020-11-17 00:00:00,␣
↪2020-11-18 00:00:00, 2020-11-19 00:00:00, 2020-11-20 00:00:00, 2020-11-21 00:00:00,␣
↪2020-11-22 00:00:00, 2020-11-23 00:00:00, 2020-11-24 00:00:00, 2020-11-25 00:00:00,␣
↪2020-11-26 00:00:00, 2020-11-27 00:00:00, 2020-11-28 00:00:00, 2020-11-29 00:00:00,␣
↪2020-11-30 00:00:00], 149180030: [2020-11-01 00:00:00, 2020-11-02 00:00:00, 2020-11-03␣
↪00:00:00, 2020-11-04 00:00:00, 2020-11-05 00:00:00, 2020-11-06 00:00:00, 2020-11-07␣
↪00:00:00, 2020-11-08 00:00:00, 2020-11-09 00:00:00, 2020-11-10 00:00:00, 2020-11-11␣
↪00:00:00, 2020-11-12 00:00:00, 2020-11-13 00:00:00, 2020-11-14 00:00:00, 2020-11-15␣
↪00:00:00, 2020-11-16 00:00:00, 2020-11-17 00:00:00, 2020-11-18 00:00:00, 2020-11-19␣
↪00:00:00, 2020-11-20 00:00:00, 2020-11-21 00:00:00, 2020-11-22 00:00:00, 2020-11-23␣
↪00:00:00, 2020-11-24 00:00:00, 2020-11-25 00:00:00, 2020-11-26 00:00:00, 2020-11-27␣
↪00:00:00, 2020-11-28 00:00:00, 2020-11-29 00:00:00, 2020-11-30 00:00:00], 149180040:␣
↪[2020-11-01 00:00:00, 2020-11-02 00:00:00, 2020-11-03 00:00:00, 2020-11-04 00:00:00,␣
↪2020-11-05 00:00:00, 2020-11-06 00:00:00, 2020-11-07 00:00:00, 2020-11-08 00:00:00,␣
↪2020-11-09 00:00:00, 2020-11-10 00:00:00, 2020-11-11 00:00:00, 2020-11-12 00:00:00,␣
↪2020-11-13 00:00:00, 2020-11-14 00:00:00, 2020-11-15 00:00:00, 2020-11-16 00:00:00,␣
↪2020-11-17 00:00:00, 2020-11-18 00:00:00, 2020-11-19 00:00:00, 2020-11-20 00:00:00,␣
↪2020-11-21 00:00:00, 2020-11-22 00:00:00, 2020-11-23 00:00:00, 2020-11-24 00:00:00,␣
↪2020-11-25 00:00:00, 2020-11-26 00:00:00, 2020-11-27 00:00:00, 2020-11-28 00:00:00,␣
↪2020-11-29 00:00:00, 2020-11-30 00:00:00], 149180050: [2020-11-01 00:00:00, 2020-11-02␣
↪00:00:00, 2020-11-03 00:00:00, 2020-11-04 00:00:00, 2020-11-05 00:00:00, 2020-11-06␣
↪00:00:00, 2020-11-07 00:00:00, 2020-11-08 00:00:00, 2020-11-09 00:00:00, 2020-11-10␣
↪00:00:00, 2020-11-11 00:00:00, 2020-11-12 00:00:00, 2020-11-13 00:00:00, 2020-11-14␣
↪00:00:00, 2020-11-15 00:00:00, 2020-11-16 00:00:00, 2020-11-17 00:00:00, 2020-11-18␣
↪00:00:00, 2020-11-19 00:00:00, 2020-11-20 00:00:00, 2020-11-21 00:00:00, 2020-11-22␣
↪00:00:00, 2020-11-23 00:00:00, 2020-11-24 00:00:00, 2020-11-25 00:00:00, 2020-11-26␣
↪00:00:00, 2020-11-27 00:00:00, 2020-11-28 00:00:00, 2020-11-29 00:00:00, 2020-11-30␣
↪00:00:00], 149180060: [2020-11-01 00:00:00, 2020-11-02 00:00:00, 2020-11-03 00:00:00,␣
↪2020-11-04 00:00:00, 2020-11-05 00:00:00, 2020-11-06 00:00:00, 2020-11-07 00:00:00,␣
↪2020-11-08 00:00:00, 2020-11-09 00:00:00, 2020-11-10 00:00:00, 2020-11-11 00:00:00,␣
↪2020-11-12 00:00:00, 2020-11-13 00:00:00, 2020-11-14 00:00:00, 2020-11-15 00:00:00,␣
↪2020-11-16 00:00:00, 2020-11-17 00:00:00, 2020-11-18 00:00:00, 2020-11-19 00:00:00,␣
↪2020-11-20 00:00:00, 2020-11-21 00:00:00, 2020-11-22 00:00:00, 2020-11-23 00:00:00,␣
↪2020-11-24 00:00:00, 2020-11-25 00:00:00, 2020-11-26 00:00:00, 2020-11-27 00:00:00,␣
↪2020-11-28 00:00:00, 2020-11-29 00:00:00, 2020-11-30 00:00:00], 149180070: [2020-11-01,␣
↪00:00:00, 2020-11-02 00:00:00, 2020-11-03 00:00:00, 2020-11-04 00:00:00, 2020-11-05,␣
↪00:00:00, 2020-11-06 00:00:00, 2020-11-07 00:00:00, 2020-11-08 00:00:00, 2020-11-09,␣
```

(continues on next page)

(continued from previous page)

```
grouper.get_group(149180020)
```

```
type(grouper.get_group(149180020))
```

```
grouper.get_group(149180020).shape
```

```
(30, 4)
```

```
grouper.get_group(149180020).head()
```

```
grouper.get_group(149180020).tail()
```

```
for name, group in grouper:
    print(name, group.shape)
    break
```

```
149180010 (30, 4)
```

```
for idx, (name, group) in enumerate(grouper):
    print(name, group.shape)
    if idx > 5:
        break
```

```
149180010 (30, 4)
149180020 (30, 4)
149180030 (30, 4)
149180040 (30, 4)
149180050 (30, 4)
149180060 (30, 4)
149180070 (30, 4)
```

```
for idx, (name, group) in enumerate(grouper):
    pass

print(idx)
```

```
839
```

What is the mean discharge for each station?

```
for idx, (name, group) in enumerate(grouper):
    print(name, group['q_cms'].mean())
    if idx > 5:
        break
```

```
149180010 25.900002
149180020 17.079998
```

(continues on next page)

(continued from previous page)

```
149180030 7.032334
149180040 0.58933336
149180050 0.6753333
149180060 3.5083337
149180070 0.15820001
```

What is the mean, min and max temperature for each station?

```
for idx, (name, group) in enumerate(grouper):
    print(name, group['temp_C'].mean(), group['temp_C'].min(), group['temp_C'].max())
    if idx > 5:
        break
```

```
149180010 99.89999 99.9 99.9
149180020 99.89999 99.9 99.9
149180030 99.89999 99.9 99.9
149180040 99.89999 99.9 99.9
149180050 99.89999 99.9 99.9
149180060 99.89999 99.9 99.9
149180070 99.89999 99.9 99.9
```

replace values > 99.8 and less than 100 with np.nan

```
df.loc[(df['temp_C']>99.8) & (df['temp_C']<100.0), 'temp_C'] = np.nan
```

```
for idx, (name, group) in enumerate(grouper):
    if not np.isnan(group['temp_C'].mean()):
      print(name, group['temp_C'].mean(), group['temp_C'].min(), group['temp_C'].max())
    if idx > 50:
        break
```

```
149180100 7.933334 5.6 10.9
149180240 5.520001 5.0 6.3
149190240 4.55 2.0 7.5
149190310 9.059999 8.2 10.6
```

Total running time of the script: (0 minutes 1.526 seconds)

5.7 multi-indexing

Important: This lesson is still under development.

Total running time of the script: (0 minutes 0.000 seconds)

5.8 efficient pandas

This file shows the how to efficiently use pandas

Important: This lesson is still under development.

```
import time
from typing import Union

import numpy as np
import pandas as pd

print(pd.__version__, np.__version__)
```

```
1.5.3 1.26.4
```

Define a function which prints memory used by a dataframe

```
def memory_usage(dataframe):
    return round(dataframe.memory_usage().sum() / 1024**2, 4)
```

don't use csv for large data

```
def dump_and_load(dataframe:pd.DataFrame):
    st = time.time()
    dataframe.to_csv("File.csv")
    pd.read_csv("File.csv")
    return round(time.time() - st, 3)

df = pd.DataFrame(np.random.random((100, 10)))

dump_and_load(df)
```

```
0.004
```

```
df = pd.DataFrame(np.random.random((1000, 20)))
dump_and_load(df)
```

```
0.032
```

```
df = pd.DataFrame(np.random.random((10_000, 50)))
dump_and_load(df)
```

```
0.779
```

```
df = pd.DataFrame(np.random.random((100_000, 50)))
dump_and_load(df)
```

```
7.517
```

```
def dump_and_load_parquet(dataframe:pd.DataFrame):

    dataframe.columns = dataframe.columns.map(str)  # parquet expects column names to be␣
↪string

    st = time.time()
    dataframe.to_parquet("File.pq")
    pd.read_parquet("File.pq")
    return round(time.time() - st, 3)

dump_and_load_parquet(df)
```

```
0.598
```

categorical type instead of string type

don't think in terms of rows, but in terms columns

```
df = pd.DataFrame(np.random.random((5000, 4)), columns=['a', 'b', 'c', 'd'])
print(df)
```

```
             a         b         c         d
0     0.929256  0.725679  0.310663  0.964345
1     0.721586  0.637415  0.332982  0.689528
2     0.933878  0.268280  0.083572  0.828697
3     0.015732  0.994194  0.238057  0.191573
4     0.509337  0.371048  0.327277  0.294127
...        ...       ...       ...       ...
4995  0.930997  0.454958  0.712097  0.878669
4996  0.640371  0.206201  0.820313  0.351462
4997  0.426621  0.693299  0.592340  0.791978
4998  0.240445  0.387177  0.034030  0.380769
4999  0.582327  0.699955  0.371898  0.463552

[5000 rows x 4 columns]
```

Iterating over rows is a lot slower than iterating over columns. This is mainly because pandas is built around column major format. This means consective values in columns are stored next to each other in memory.

```
start = time.time()
for col in df.columns:
    for val in df[col]:
        pass
print(time.time() - start)
```

```
0.002445697784423828
```

```
start = time.time()
for row_idx in range(len(df)):
    for val in df.iloc[row_idx]:
        pass
print(time.time() - start)
```

```
0.17723488807678223
```

```
start = time.time()
for idx, i in enumerate(range(len(df))):
    row = df.iloc[idx]
    row.loc['a'] = row.loc['a'] + row.loc['b']
    df.iloc[idx] = row
print(time.time() - start)
```

```
0.8116934299468994
```

```
print(df)
```

```
             a         b         c         d
0     1.654934  0.725679  0.310663  0.964345
1     1.359001  0.637415  0.332982  0.689528
2     1.202158  0.268280  0.083572  0.828697
3     1.009926  0.994194  0.238057  0.191573
4     0.880384  0.371048  0.327277  0.294127
...        ...       ...       ...       ...
4995  1.385956  0.454958  0.712097  0.878669
4996  0.846572  0.206201  0.820313  0.351462
4997  1.119921  0.693299  0.592340  0.791978
4998  0.627622  0.387177  0.034030  0.380769
4999  1.282282  0.699955  0.371898  0.463552

[5000 rows x 4 columns]
```

```
start = time.time()
df['a'] = df['a'] + df['b']
print(time.time() - start)
```

```
0.0006124973297119141
```

Use vectorized operations instead of iterating or using `apply` method

Use chaining instead of creating new dataframes after every operation

reduce memory consuption

Let's create a dataframe with column which contains only integers

```
df = pd.DataFrame(np.random.randint(0, 256, 10000000))

print(df.dtypes)
```

```
0    int64
dtype: object
```

The default type fo the column is `object` which means pandas does not know that the data in column is only integer.

The memory consumed by the dataframe currently is:

```
print(f"{memory_usage(df)} Mb")
```

```
76.2941 Mb
```

However when we check the maximum and minimum value of integers in our dataframe they range between 0 and 255.

```
print(df[0].min(), df[0].max())
```

```
0 255
```

This means we can store our data as int16. With `object` type, we are assigning a lot of memory to our data, which is even not necessary.

We can verify that the maximum and minium value in the column is between the lower and upper limit of of np.int16.

```
print(df[0].min() > np.iinfo(np.int16).min and df[0].max() < np.iinfo(np.int16).max)
```

```
True
```

So now let's convert our the data type of our column into np.int16 and check the memory consuption now.

```
df[0] = df[0].astype(np.int16)

print(f"{memory_usage(df)} Mb")
```

```
19.0736 Mb
```

we see the memory usage has been reduced significantly. Now let's do same with floats.

```
df = pd.DataFrame(np.random.random(10000000))

print(df.dtypes)

print(f"Initial memory: {memory_usage(df)} Mb")

print(f"min: {df[0].min()} max:  {df[0].max()}")

print(df[0].min() > np.iinfo(np.int16).min and df[0].max() < np.iinfo(np.int16).max)

```

(continues on next page)

(continued from previous page)

```
df[0] = df[0].astype(np.float16)

print(f"Final memory:  {memory_usage(df)} Mb")
```

```
0    float64
dtype: object
Initial memory: 76.2941 Mb
min: 2.4135430432004057e-07 max:  0.9999999696427855
True
Final memory:  19.0736 Mb
```

We can write helper functions to convert the column types in dataframe. Below, we write functions, which check the data in each column of a dataframe, and assign the the dtype (read as assign the memory) which is just enough for the data in column. It means we assign the memory enough for the column but not more than what is required.

```
def int8(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.iinfo(np.int8).min and array.max() < np.iinfo(np.int8).max

def int16(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.iinfo(np.int16).min and array.max() < np.iinfo(np.int16).max

def int32(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.iinfo(np.int32).min and array.max() < np.iinfo(np.int32).max

def int64(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.iinfo(np.int64).min and array.max() < np.iinfo(np.int64).max

def float16(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.finfo(np.float16).min and array.max() < np.finfo(np.float16).
↪max

def float32(array:Union[np.ndarray, pd.Series])->bool:
    return array.min() > np.finfo(np.float32).min and array.max() < np.finfo(np.float32).
↪max


def maybe_convert_int(series:pd.Series)->pd.Series:
    if int8(series):
        return series.astype(np.int8)
    if int16(series):
        return series.astype(np.int16)
    if int32(series):
        return series.astype(np.int32)
    if int64(series):
        return series.astype(np.int64)
    return series


def maybe_convert_float(series:pd.Series)->pd.Series:

    if float16(series):
        return series.astype(np.float16)
```

(continues on next page)

(continued from previous page)

```
    if float32(series):
        return series.astype(np.float32)

    return series


def maybe_reduce_memory(dataframe:pd.DataFrame, hints=None)->pd.DataFrame:

    init_memory = memory_usage(dataframe)

    if hints:
        assert len(hints) == len(dataframe.columns)
    else:
        hints = {col:'' for col in dataframe.columns}

    for col in dataframe.columns:
        col_dtype = str(dataframe[col].dtypes)

        if col_dtype in  ['int8', 'int16', 'int32', 'int64', 'float16', 'float32',
↪'float64']:

            if 'int' in  hints[col]:
                dataframe[col] = maybe_convert_int(dataframe[col])
            elif 'float' in hints[col]:
                dataframe[col] = maybe_convert_float(dataframe[col])
            elif 'int' in col_dtype:
                dataframe[col] = maybe_convert_int(dataframe[col])
            elif 'float' in col_dtype or 'float' in  hints[col]:
                dataframe[col] = maybe_convert_float(dataframe[col])

    print(f"memory reduced from {init_memory} to {memory_usage(dataframe)}")
    return dataframe
```

Now we can test our function that how much memory it reduces.

```
df = pd.DataFrame(np.column_stack([
    np.random.randint(-126, 126, 100_000),
    np.random.randint(-31000, 32760, 100_000),
    np.random.randint(0, 2147483640, 100_000),
]))

print(df.shape)
```

```
(100000, 3)
```

Print the original dtypes

```
print(df.dtypes)
```

```
0    int64
1    int64
2    int64
```

(continues on next page)

(continued from previous page)

```
dtype: object
```

```
maybe_reduce_memory(df)
```

```
memory reduced from 2.2889 to 0.6677
```

print the converted dtypes

```
print(df.dtypes)
```

```
0     int8
1    int16
2    int32
dtype: object
```

Test with dataframe containing floats

```
df = pd.DataFrame(np.column_stack([
    np.random.randint(-126, 65000, 100_000) * 1.0,
    np.random.randint(-31000, 100_000, 100_000)*1.0,
]))

print(df.dtypes)
maybe_reduce_memory(df)
print(df.dtypes)
```

```
0    float64
1    float64
dtype: object
memory reduced from 1.526 to 0.5723
0    float16
1    float32
dtype: object
```

```
df = pd.DataFrame(np.column_stack([
    np.random.randint(-126, 126, 100_000),
    np.random.randint(-31000, 32760, 100_000),
    np.random.randint(0, 2147483640, 100_000),
    np.random.randint(-126, 65000, 100_000) * 1.0,
    np.random.randint(-31000, 100_000, 100_000)*1.0,
]))

print(df.dtypes)
maybe_reduce_memory(df)
print(df.dtypes)
```

```
0    float64
1    float64
2    float64
3    float64
4    float64
```

(continues on next page)

(continued from previous page)

```
dtype: object
memory reduced from 3.8148 to 1.3353
0    float16
1    float16
2    float32
3    float16
4    float32
dtype: object
```

```
df = pd.DataFrame(np.column_stack([
    np.random.randint(-126, 126, 100_000),
    np.random.randint(-31000, 32760, 100_000),
    np.random.randint(0, 2147483640, 100_000),
    np.random.randint(-126, 65000, 100_000) * 1.0,
    np.random.randint(-31000, 100_000, 100_000)*1.0,
]))

print(df.dtypes)
maybe_reduce_memory(df, hints={0: "int", 1: "int", 2: "int",
                               3: "float", 4: "float"})
print(df.dtypes)
```

```
0    float64
1    float64
2    float64
3    float64
4    float64
dtype: object
memory reduced from 3.8148 to 1.2399
0       int8
1      int16
2      int32
3    float16
4    float32
dtype: object
```

For smaller dataframes, teh differene may not seem much but when we try to scale things up, the difference is very significant

```
df = pd.DataFrame(np.column_stack([
    np.random.randint(-126, 126, 1000_000),
    np.random.randint(-31000, 32760, 1000_000),
    np.random.randint(0, 2147483640, 1000_000),
    np.random.randint(-126, 65000, 1000_000) * 1.0,
    np.random.randint(-31000, 100_000, 1000_000)*1.0,
]))

print(df.dtypes)
maybe_reduce_memory(df, hints={0: "int", 1: "int", 2: "int",
                               3: "float", 4: "float"})
print(df.dtypes)
```

```
0    float64
1    float64
2    float64
3    float64
4    float64
dtype: object
memory reduced from 38.1471 to 12.3979
0       int8
1      int16
2      int32
3    float16
4    float32
dtype: object
```

References

Total running time of the script: (0 minutes 10.524 seconds)

5.9 pivot vs melt

```
import time
import pandas as pd

print(time.asctime())
print(pd.__version__)
```

```
Mon Nov 11 19:32:40 2024
1.5.3
```

Let's consider a dataframe which consists of daily streamflow data from 840 polish stations for the month of January 2020. The data is available in a zip file. We can read the data using the *read_csv* method.

```
url = "https://danepubliczne.imgw.pl/data/dane_pomiarowo_obserwacyjne/dane_hydrologiczne/
↪dobowe/2020/codz_2020_01.zip"

df = pd.read_csv(
    url, compression='zip', encoding="ISO-8859-1",  engine='python',
    on_bad_lines="skip",
    names=['stn_id', 'year', 'day', 'q_cms', 'month'],
    usecols=[0, 3, 5, 7, 9],
    dtype={'stn_id': 'str', 'year': 'int', 'day': 'int', 'q_cms': 'float', 'month': 'int
↪'},
    parse_dates={'date': ['year', 'month', 'day']},
    index_col='date'
    )


df
```

```
print(df.shape)
```

```
(24913, 2)
```

The above dataframe consists of data for 480 stations, each stacked on top of the other.

```
len(df['stn_id'].unique())
```

```
840
```

If we want data for each station in a separate column, we can use the *pivot_table* method.

```
pivoted_table = df.pivot_table(index=df.index, columns="stn_id", values="q_cms")

pivoted_table
```

```
pivoted_table.shape
```

```
(30, 840)
```

```
pivoted_table.columns
```

```
Index([' 149180010', ' 149180020', ' 149180030', ' 149180040', ' 149180050',
       ' 149180060', ' 149180070', ' 149180080', ' 149180090', ' 149180100',
       ...
       ' 154220060', ' 154220070', ' 154220080', ' 154220090', ' 154220100',
       ' 154220110', ' 154230010', ' 154230020', ' 154230030', ' 154230040'],
      dtype='object', name='stn_id', length=840)
```

```
len(pivoted_table.columns)
```

```
840
```

Melt

Melt is kind of opposite to that of pivot. It stacks the columns on top of each other.

```
melted_table = df.melt(id_vars=["stn_id"], value_vars=["q_cms"])

melted_table
```

Total running time of the script: (0 minutes 0.915 seconds)

2.7.6 6. plotting

This chapter contains lessons regarding plotting.

6.0 Introduction

This lesson introduces basics of plotting in python.

Pure python does not contain a built-in module for plotting. Fortunately, other people have developed many awesome tools for plotting in python. However, since these tools do not come together with python, it means we have to install them. One such tool is matplotlib. Once the matplotlib is installed, we can import it as below:

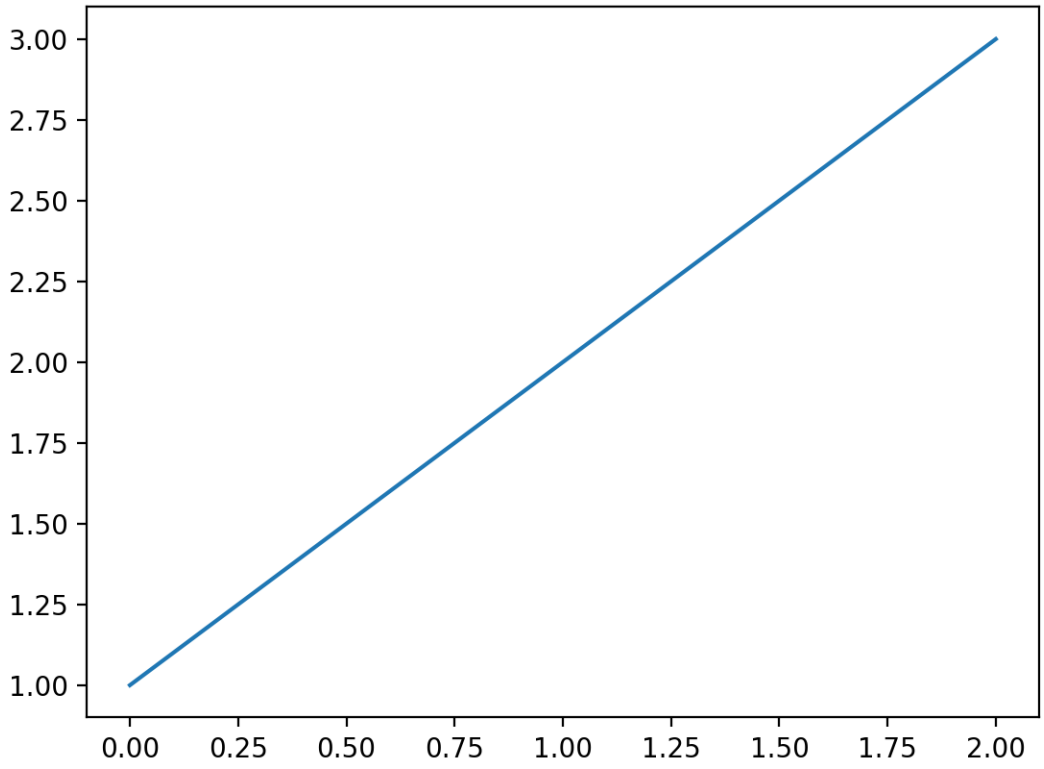
```
import matplotlib
```

the pyplot sub-module of matplotlib as

```
import matplotlib.pyplot as plt
```

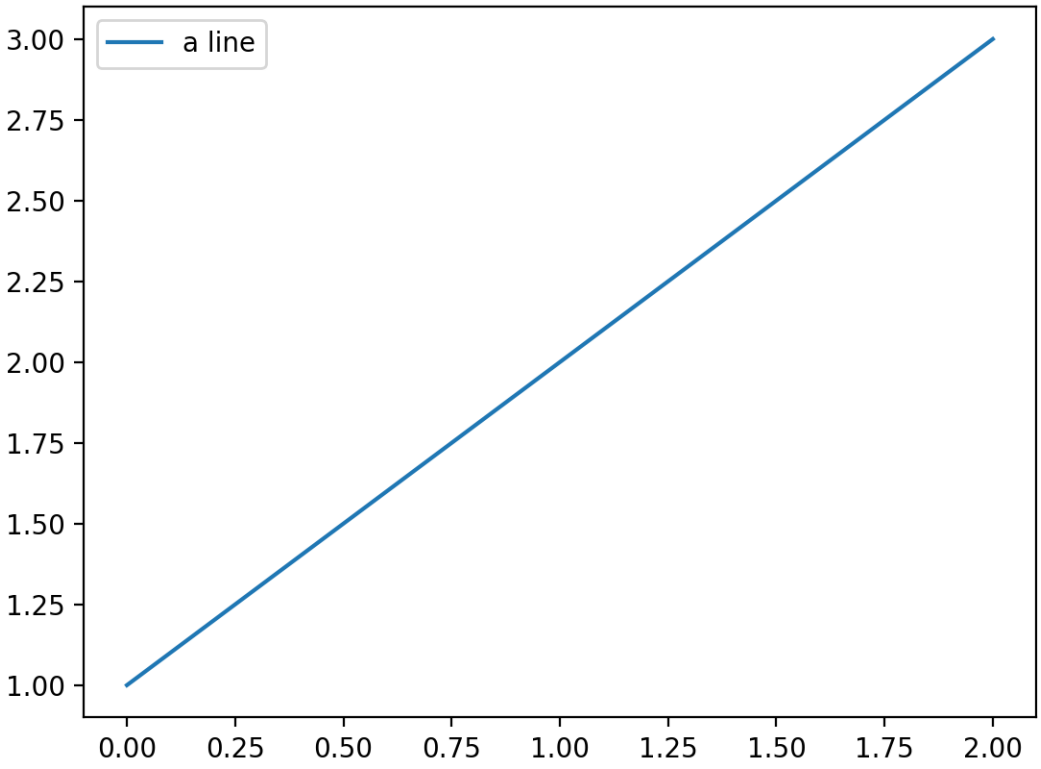
It was developed to replicate MATLAB in python. It is a very comprehensive and probably the most widely used plotting library in python. matplotlib provides great flexibility to the user to accomplish data visualization in python. However, this flexibility comes with a price. Suppose if you want to build a house, you have a choice either to make walls, doors windows, etc yourself or you can get these ready-to-use materials and then join them together. Using the first approach is more cumbersome and time consuming because you will only have raw material (wood, rocks, soil, water) to start with. However, it provides you the flexibility which you will not have if you use the second approach. The same is true for matplotlib. There are two APIs in matplotlib, the functional and object oriented. Following example shows drawing a plot using object oriented API of matplotlib.

```
fig, axes = plt.subplots()
axes.plot([1,2,3])
plt.show()
```



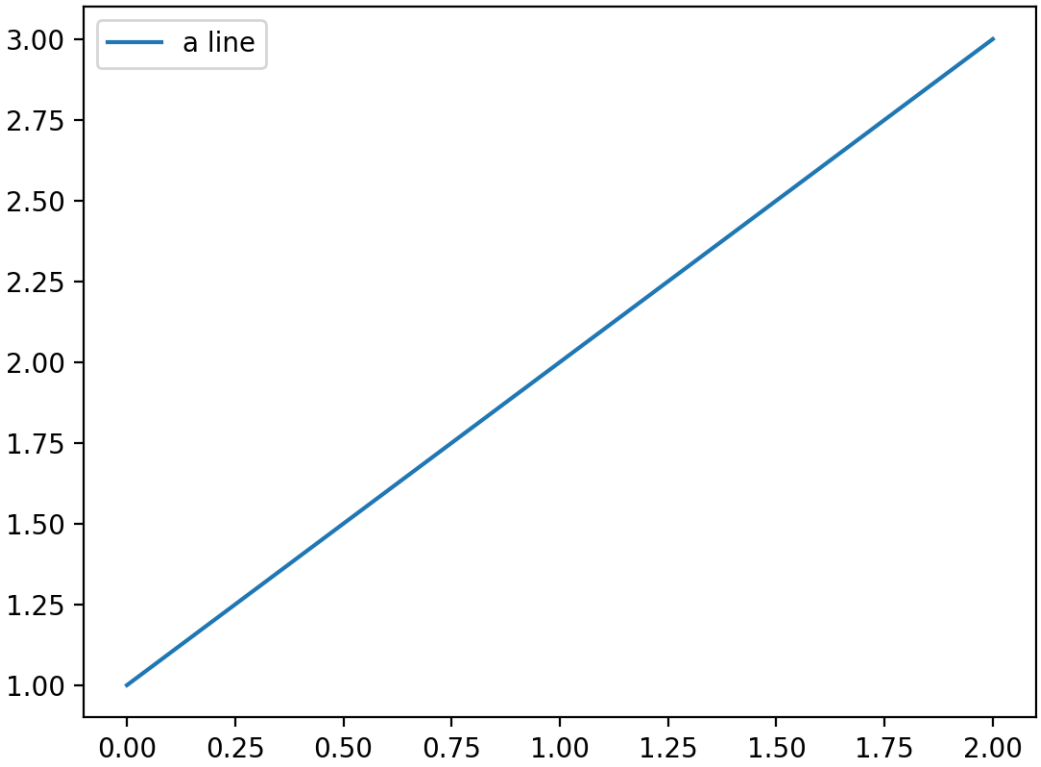
If we want to add the legend, we have to add another line as below

```
_, axes = plt.subplots()
axes.plot([1,2,3], label="a line")
plt.legend()
plt.show()
```



Above we first cretae the axes and then use it plot the list of values. We have to call `plt.show()` to see the plot. Once the plot is shown, the axes is exhausted. This means we can not work on it any further. We have to create a new axes if we want to plot a new line. Above we have created the plot using objected oriented API of matplotlib. There is also a functional API in matplotlib. To accomplish above using functional API, we can do as below

```
plt.plot([1,2,3], label="a line")
plt.legend()
plt.show()
```



As we can notice that, the functional API will fall short to our requirements most of the times. But the objected oriented API seems too verbose. What if we could do

```
plot([1,2,3], label="a line")
```

and we would be able to see the plot. This necessity gave rise to easy_mpl library. The purpose of easy_mpl is to ease the use of matplotlib while keeping the flexibility of object oriented programming paradigm of matplotlib intact. Using these one liners will save the time and will not hurt. Moreover, you can swap most function of this library with that of matplotlib and vice versa. In this chapter, we will mainly use easy_mpl library for plotting.

`easy_mpl` contains two kinds of functions, one which are just wrappers around their matplotlib alternatives. These include `plot()`, `scatter()`, `bar_chart()`, `pie()`, `hist()`, `imshow()` and `boxplot()`. As the name suggests, these are just alternatives to their matplotlib aliases. All of these functions take same input arguments as taken by corresponding matplotlib functions. If these functions are given same input arguments as to their matplotlib alternatives, then these functions return same output as returned by matplotlib. Therefore, we can consider them as alternative to matplotlib (for most cases). All these functions take three further input arguments. These are `ax`, `ax_kws` and `show`. The meanings of these three arguments are as below

- `ax` stands for axes, the matplotlib axes object `matplotlib.axes`. If `ax` argument is given, then the plots are drawn on this otherwise either a new matplotlib axes is created or currently available axes is used.
- `ax_kws` is a dictionary which includes the arguments to manipulate the x and y labels, ticklabels, title. These arguments are passed to `easy_mpl.utils.process_axes()` function.
- The `show` argument determines whether to draw the plot after the function or not. If `show` is set to False, then the axes is not *exhausted*, which means, we can manipulate it if required and call *plt.show* or *plt.draw* after

manipulating the axes. Otherwise, in default case (when `show` is True), the plot is drawn immediately after calling the corresponding function.

Moreover these wrapper functions also take some auxiliary input arguments which can be used for further manipulation of these plots. For example the `imshow()` function takes the `grid_params` argument. The second kinds of functions in this library are helper functions for data visualization and analysis. These include `regplot()`, `dumbbell_plot()`, `ridge()`, `parallel_coordinates()`, `taylor_plot()`, `lollipop_plot()`, `circular_bar_plot()`, `violin_plot()` and `spider_plot()` . Thus easy_mpl is not a replacement to matplotlib in all the cases but it can be your go to tool for the plots given in examples and API in most of the cases.

Although we will use easy_mpl in this chapter but some concepts of matplotlib will be introduced along the way. Each chapter contains comprehensive list of examples so that they can be adopted by the user. For complex examples, the user will notice switching to matplotlib by using native matplotlibl functions directly.

Total running time of the script: (0 minutes 0.831 seconds)

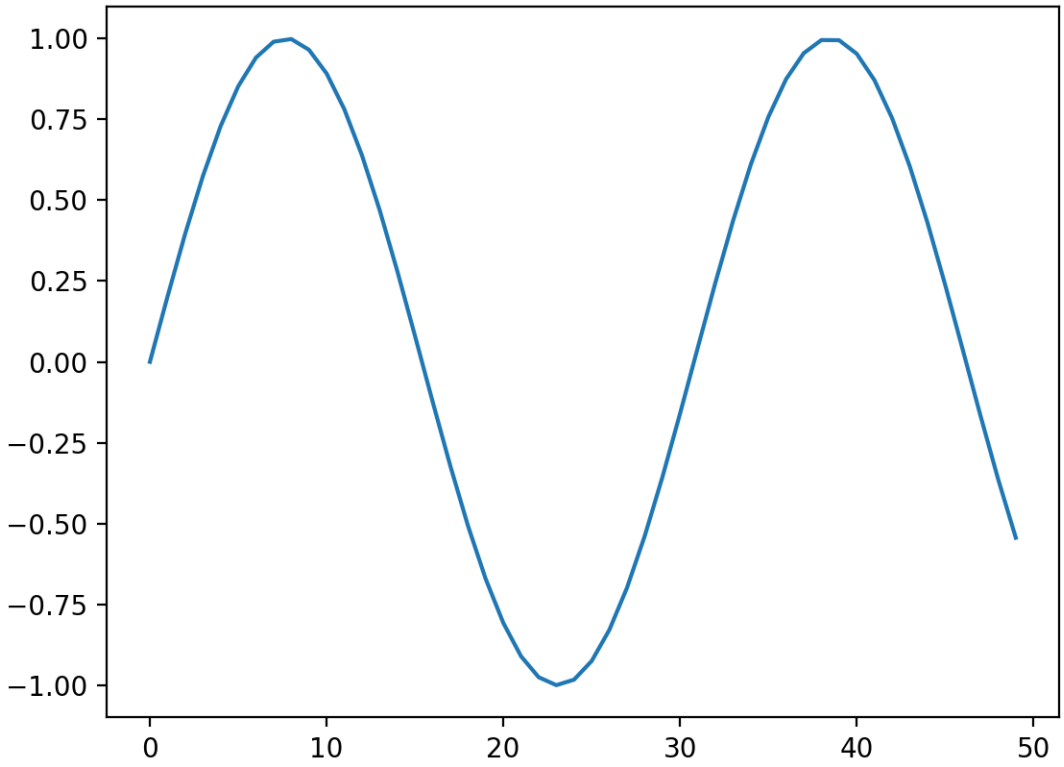
6.1 basic plot

```
# sphinx_gallery_thumbnail_number = -1

import numpy as np
import pandas as pd
from easy_mpl import plot
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from easy_mpl.utils import version_info, despine_axes

version_info()  # print version information of all the packages being used
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

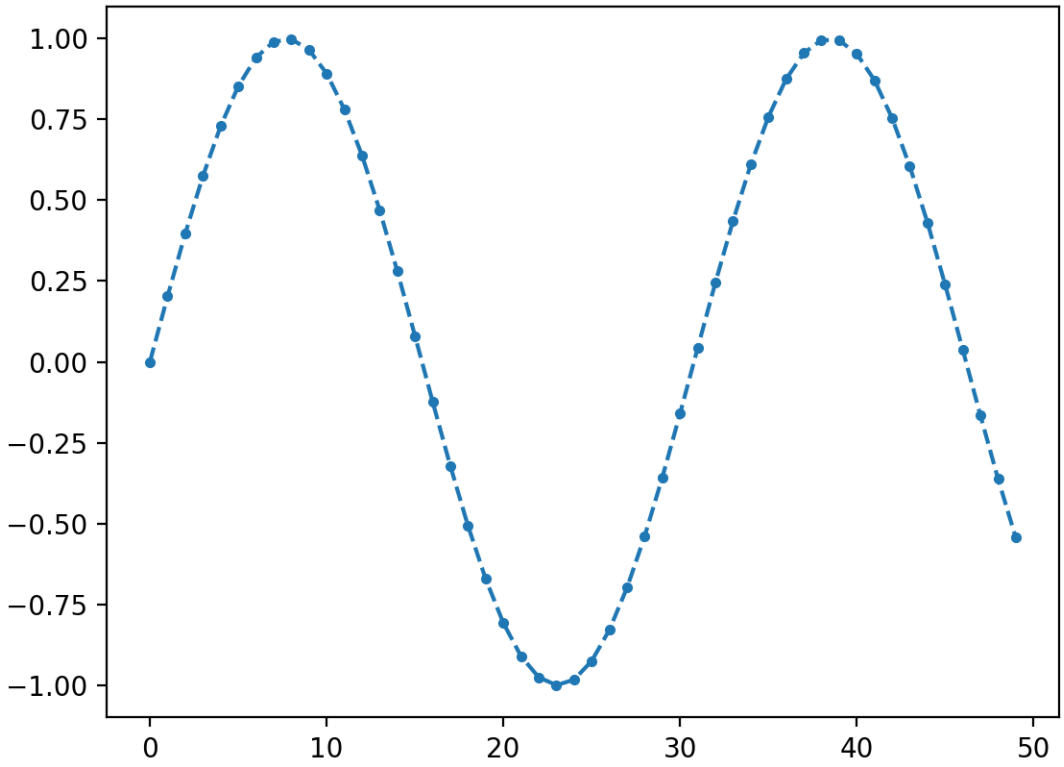
A basic plot can be drawn just by providing a sequence of numbers to the `plot` function.

```
x = np.linspace(0, 10, 50)
y = np.sin(x)
_ = plot(y)
```



We can however set the style of the plot/marker using the second argument.

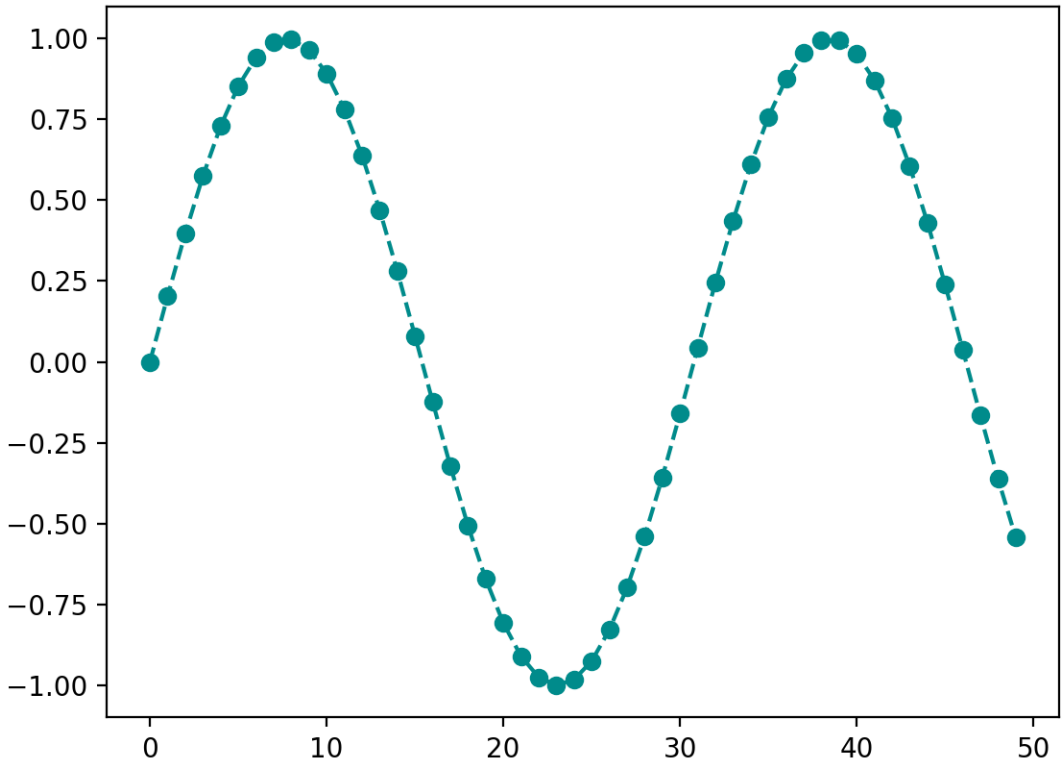
```
_ = plot(y, '--.')
```



The complete list of available marker styles can be seen *here <https://matplotlib.org/stable/api/markers_api.html>_*

The color can be specified by making use of `c` or `color` argument to `plot` function.

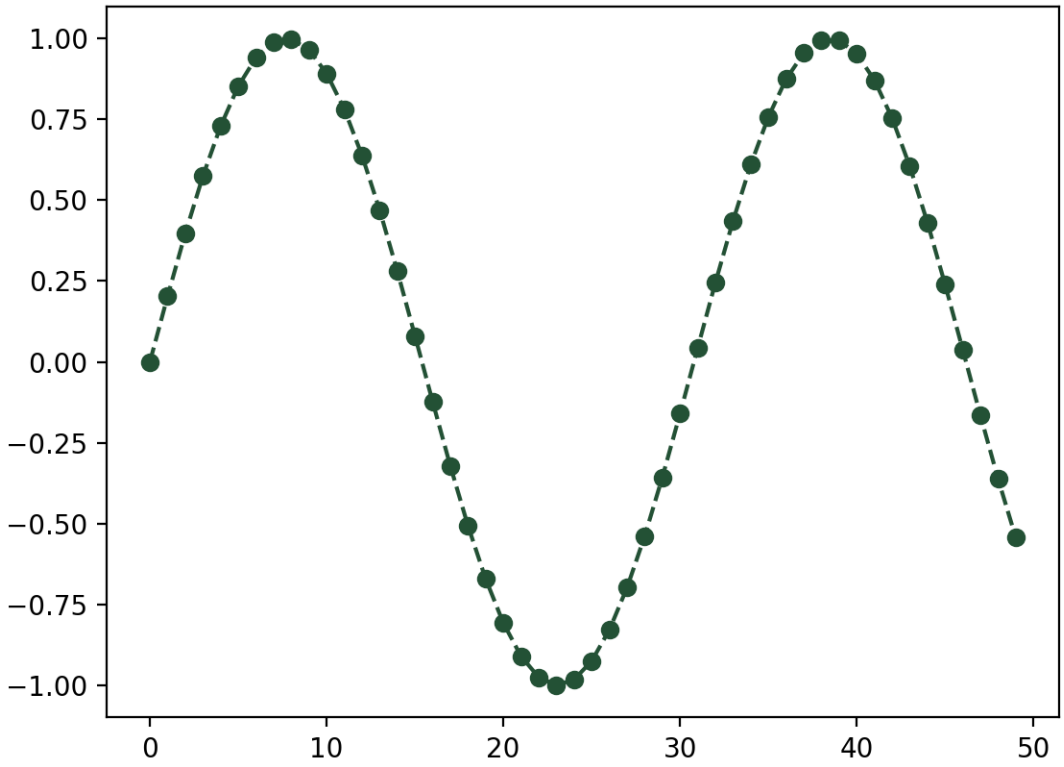
```
_ = plot(y, '--o', color='darkcyan')
```



You can refer to this page to see names of all valid matplotlib color names.

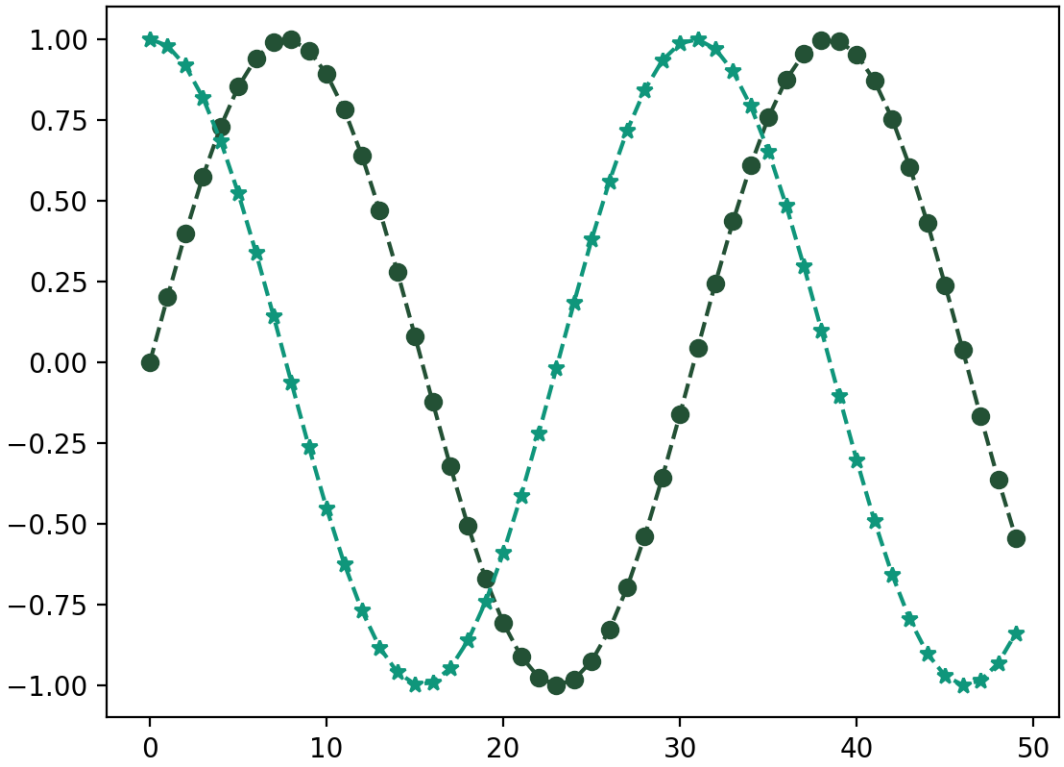
We can explicitly provide rgb values of a color.

```
_ = plot(y, '--o', color=np.array([35, 81, 53]) / 256.0)
```



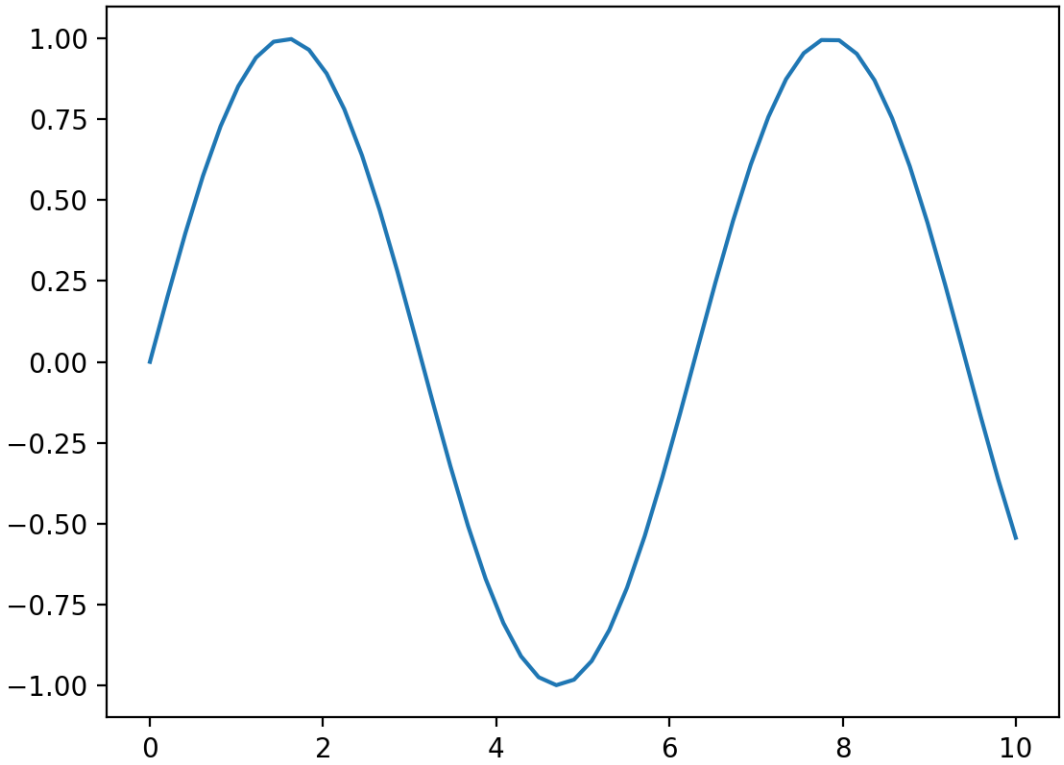
We can set the show=False in order to further work the current active axes

```
y2 = np.cos(x)
plot(y, '--o', color=np.array([35, 81, 53]) / 256.0, show=False)
_ = plot(y2, '--*', color=np.array([15, 151, 123]) / 256.0)
```



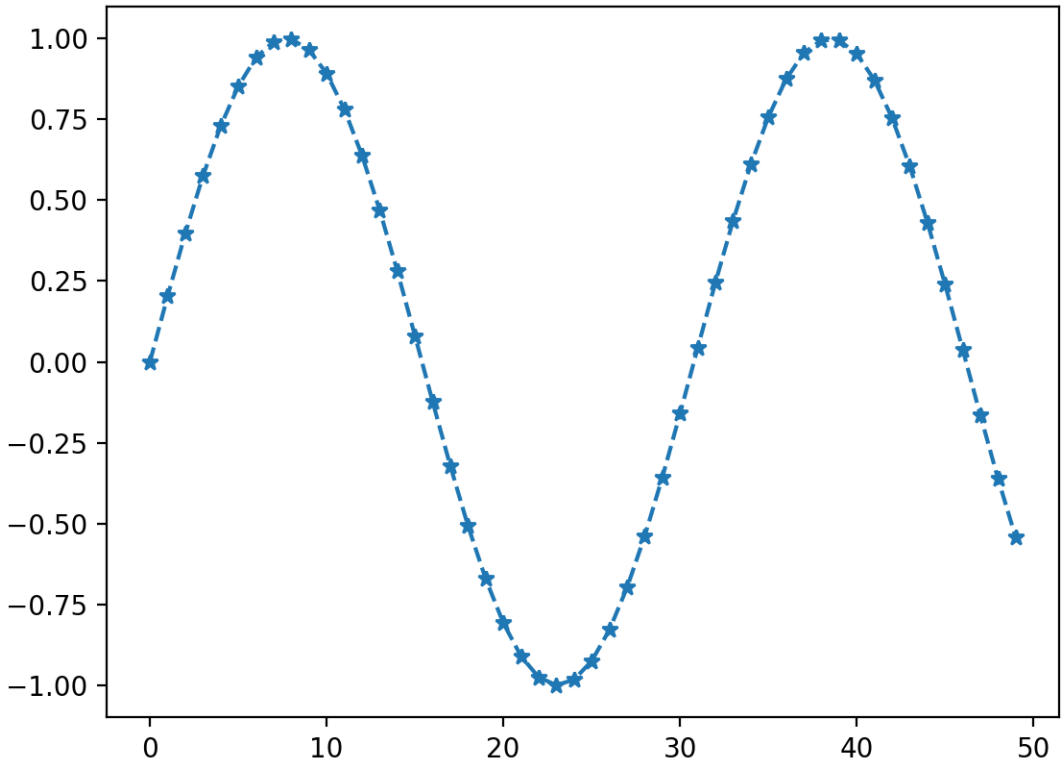
If we provide two arrays to `plot`, the first array is used for the horizontal axis and second argument/array is used as corresponding y values. In this case, the second argument is not used to define marker style.

```
_ = plot(x, y)
```



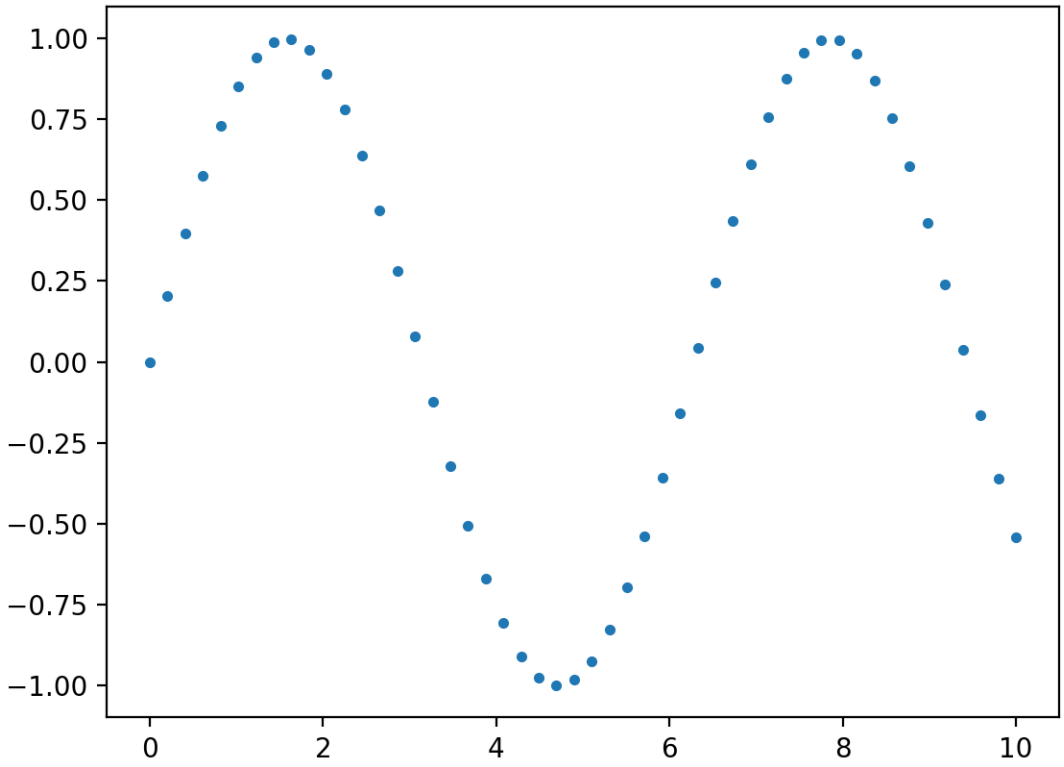
However, when we give just one array, the second argument is interpreted as marker style.

```
_ = plot(y, '--*')
```



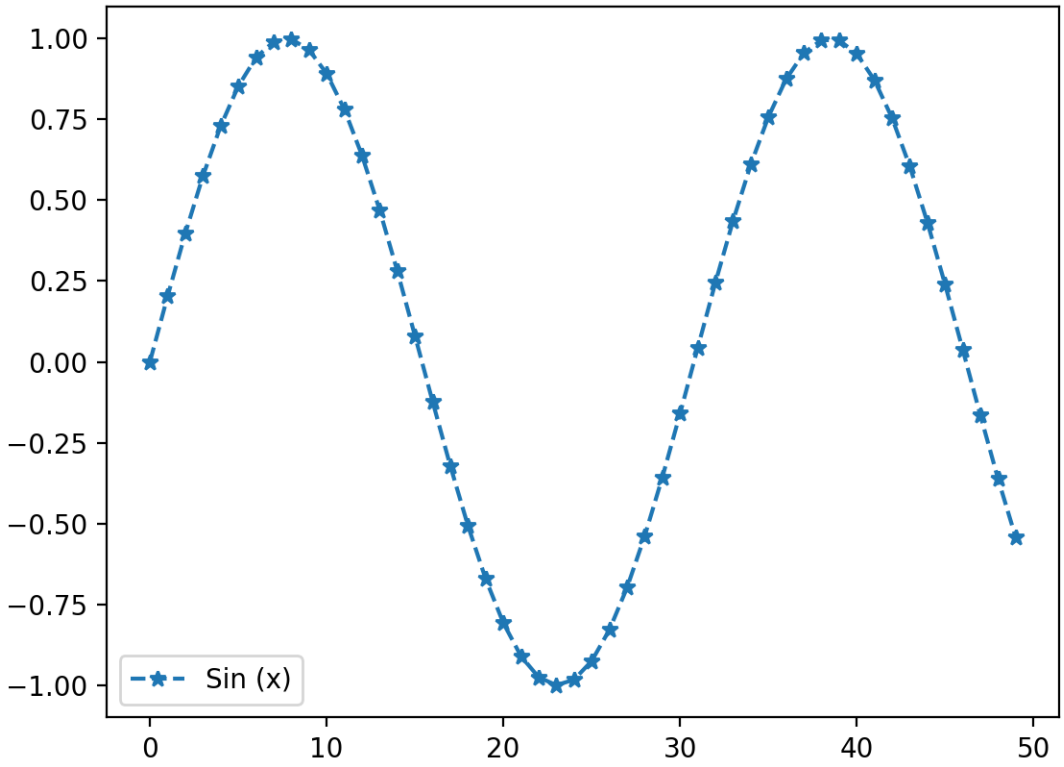
When we provide two arrays, the third argument is interpreted as marker style.

```
_ = plot(x, y, '.')
```



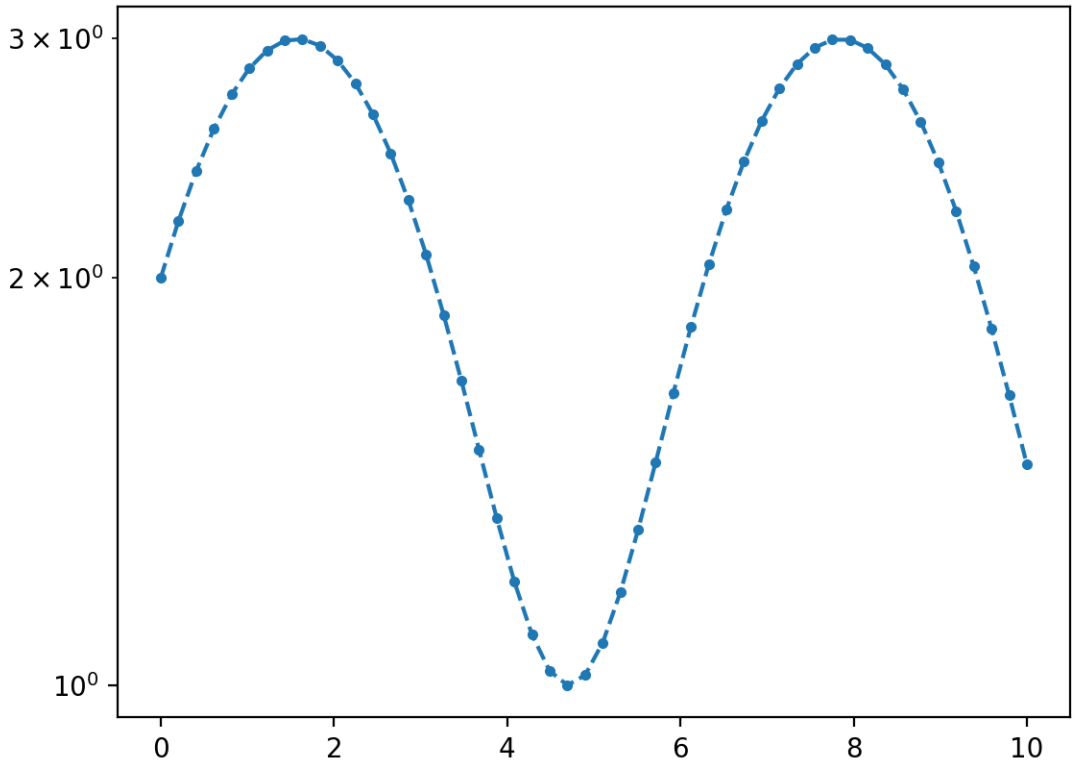
The legend can be set by making use of `label` argument.

```
_ = plot(y, '--*', label="Sin (x)")
```



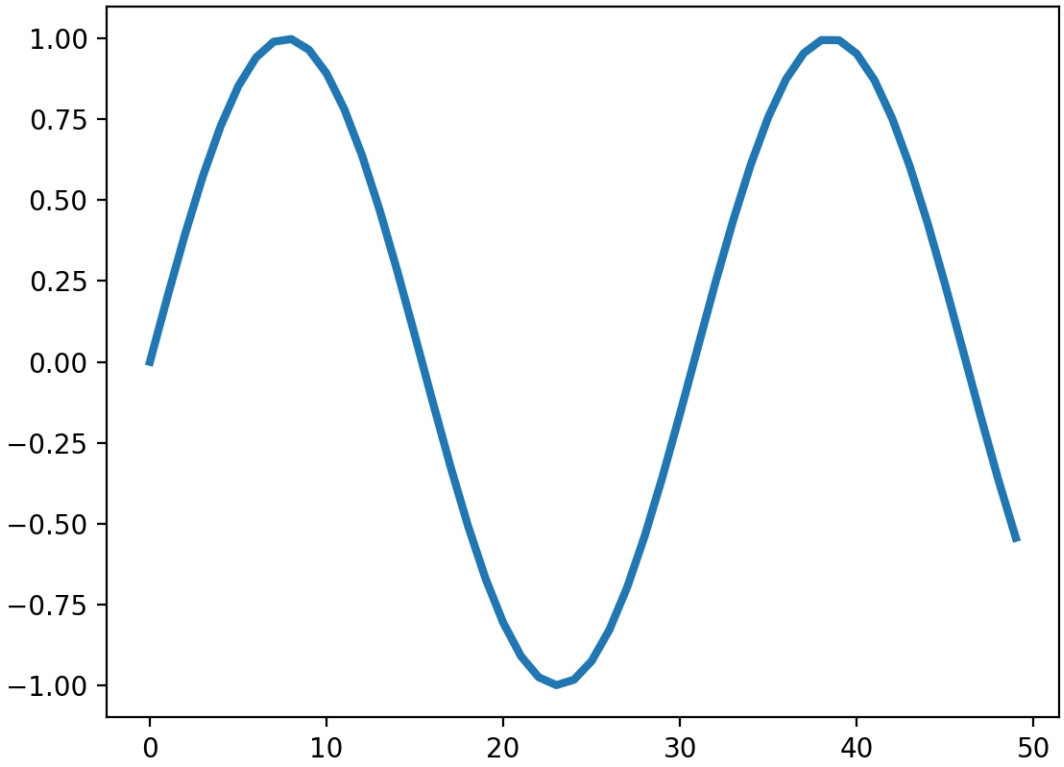
If we want the y-axis to be on log scale, we can set `logy` to True and pass it as `ax_kws` dictionary.

```
_ = plot(x, y+2, '--.',  ax_kws={'logy':True})
```

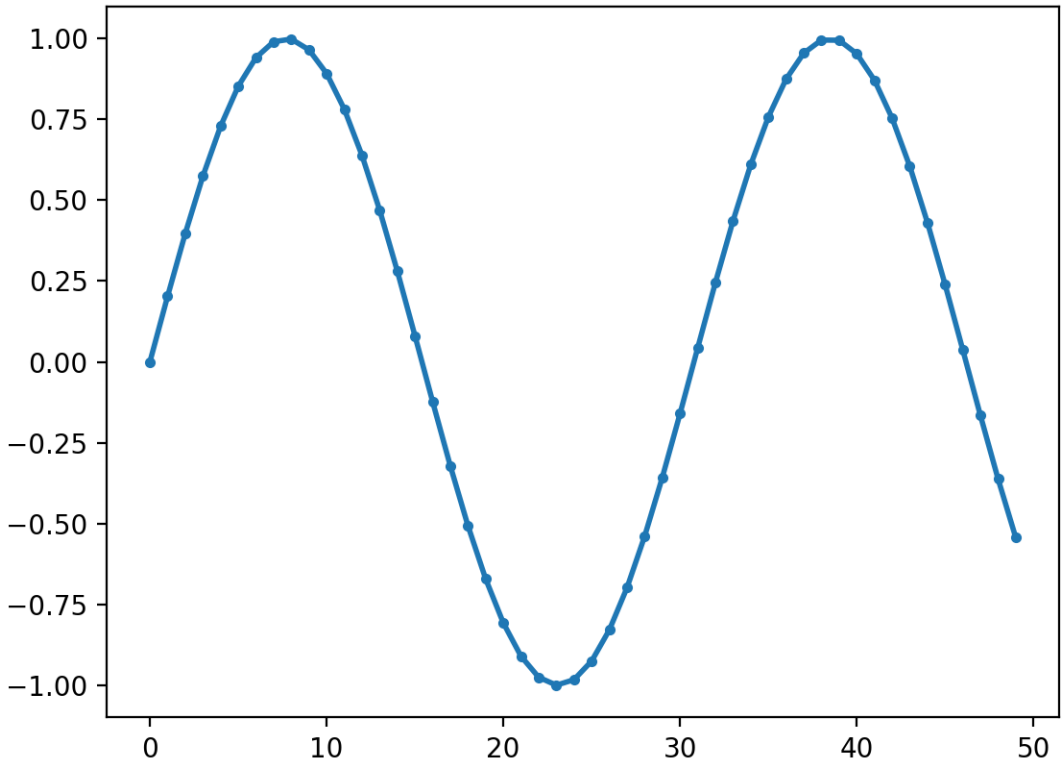


The width of the line can be set using `lw` or `linewidth` argument.

```
_ = plot(y, linewidth=3.)
```

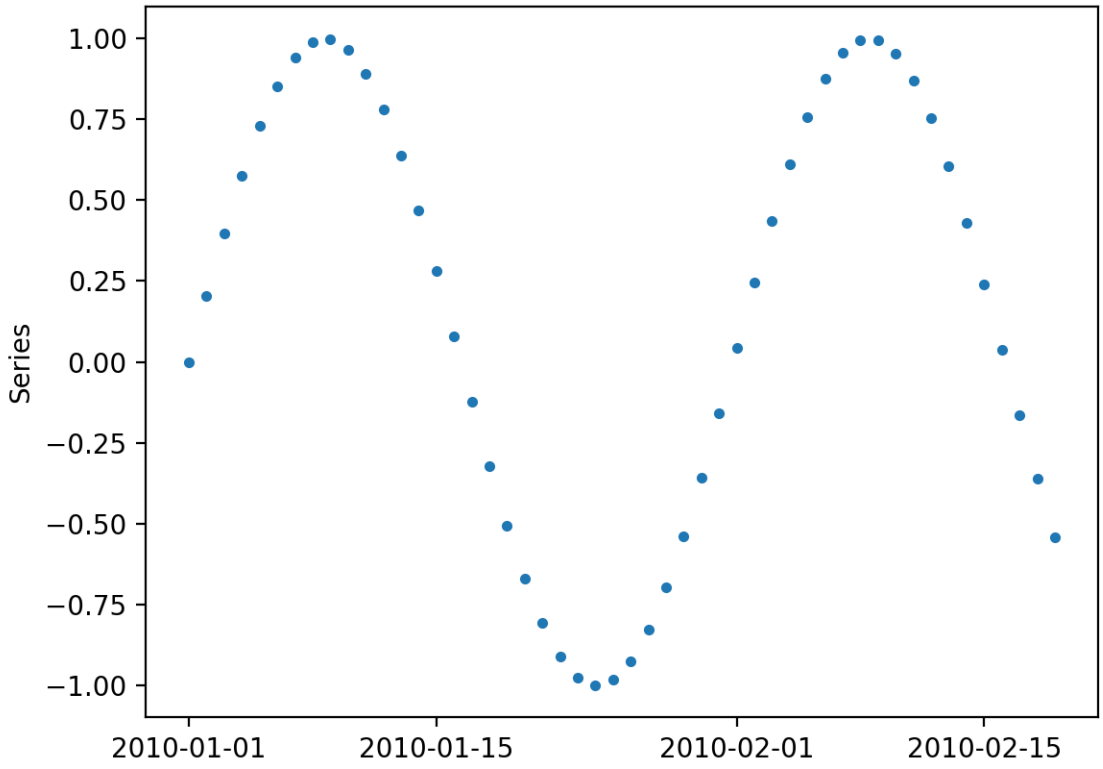



```
_ = plot(y, marker=".", lw=2)
```



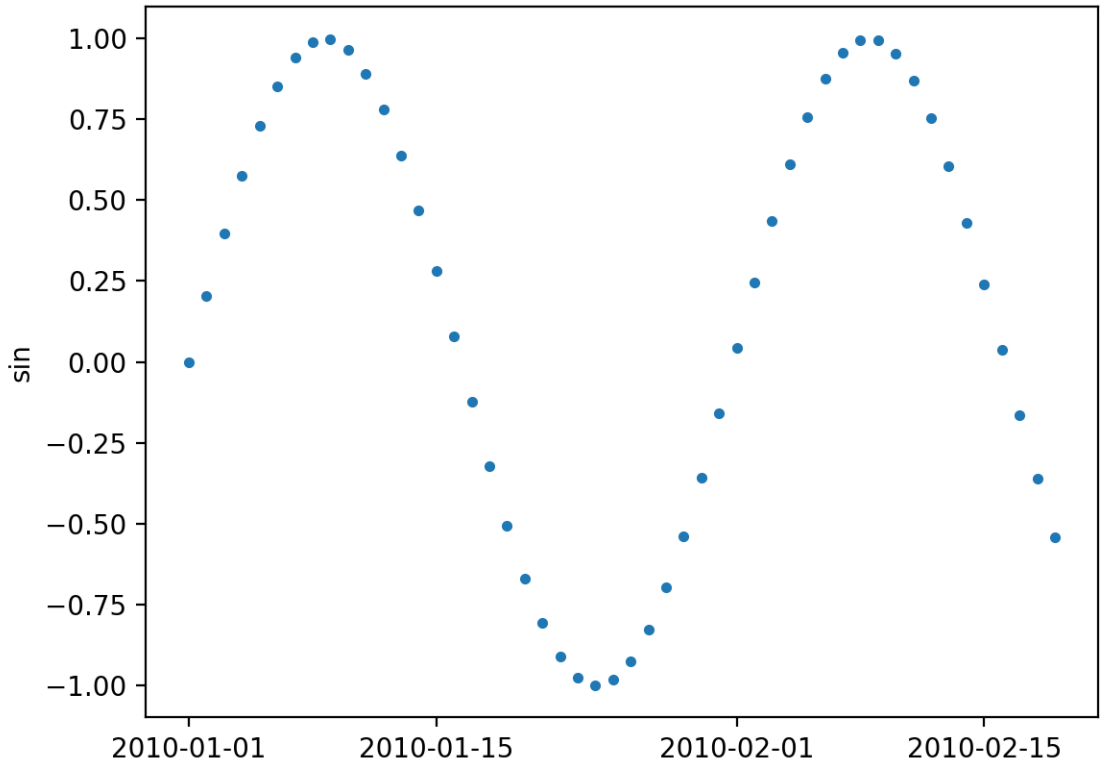
Instead of numpy array, we can also provide pandas Series

```
x = pd.Series(y, name="Series",
              index=pd.date_range("20100101", periods=len(y), freq="D"))
_ = plot(x, '.')
```



or a pandas DataFrame with 1 column

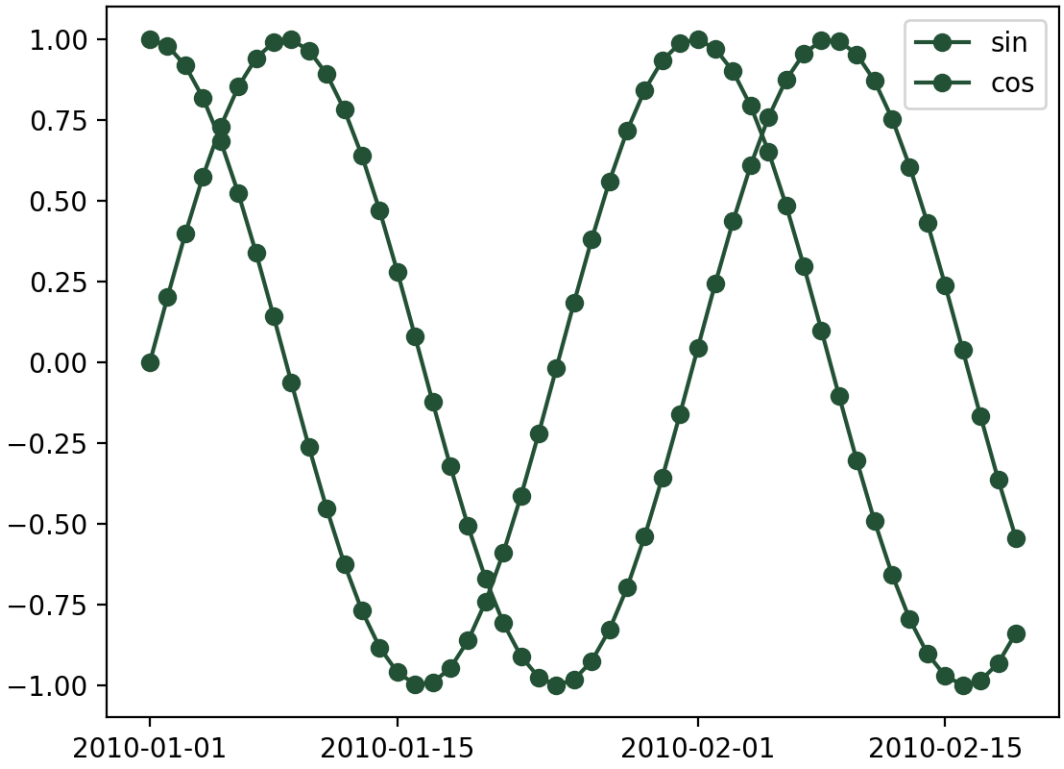
```
x = pd.DataFrame(y, columns=["sin"],
                 index=pd.date_range("20100101", periods=len(y), freq="D"))
_ = plot(x, '.')
```



It should be noted that the index of pandas Series or DataFrame, which is a DateTimeIndex in this case, is used for x-axis

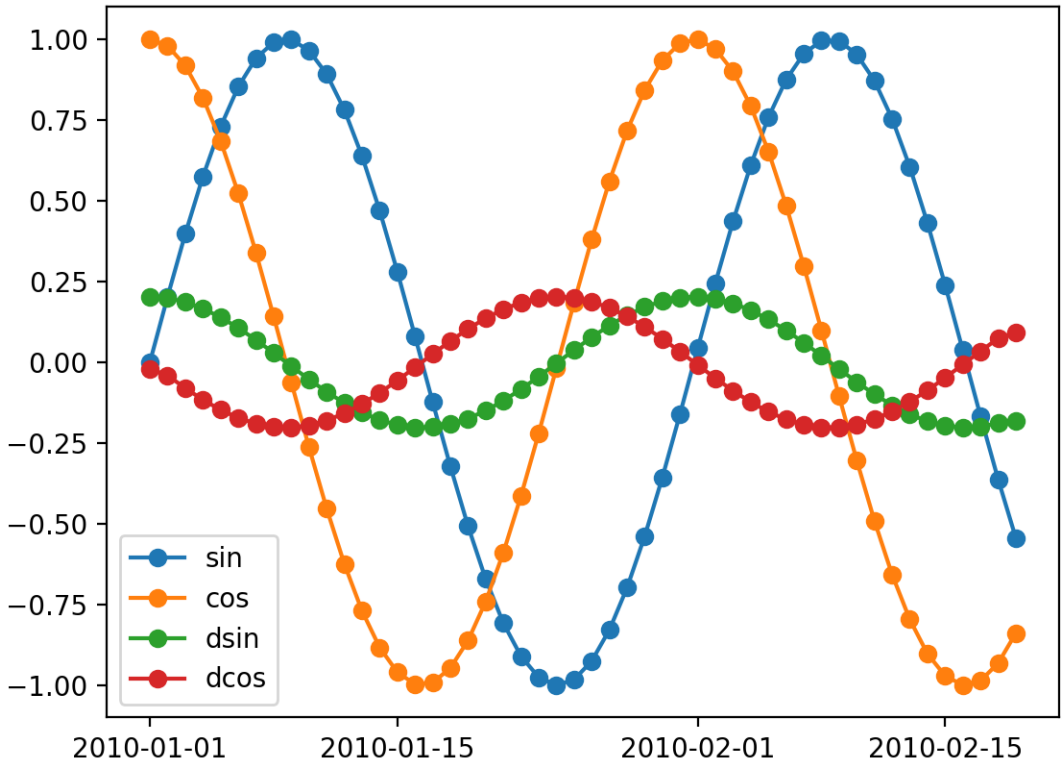
If we provide pandas DataFrame with two columns, both columns are plotted.

```
x = pd.DataFrame(np.column_stack([y, y2]),
                 columns=["sin", "cos"],
                 index=pd.date_range("20100101", periods=len(y), freq="D"))
_ = plot(x, '-o', color=np.array([35, 81, 53]) / 256.0)
```



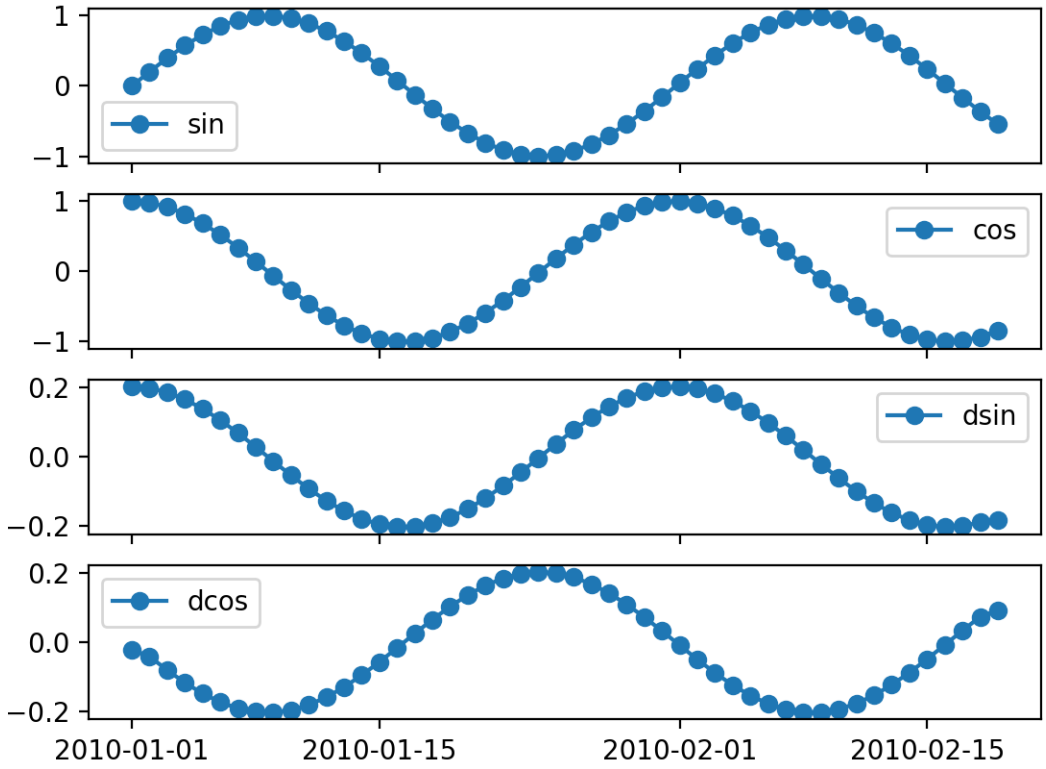
For more than one columns, if we don't fix the color, the colors are chosen randomly.

```
dy = np.gradient(y)
dy2 = np.gradient(y2)
x = pd.DataFrame(np.column_stack([y, y2, dy, dy2]),
                 columns=["sin", "cos", "dsin", "dcos"],
                 index=pd.date_range("20100101", periods=len(y), freq="D"))
_ = plot(x, '-o')
```



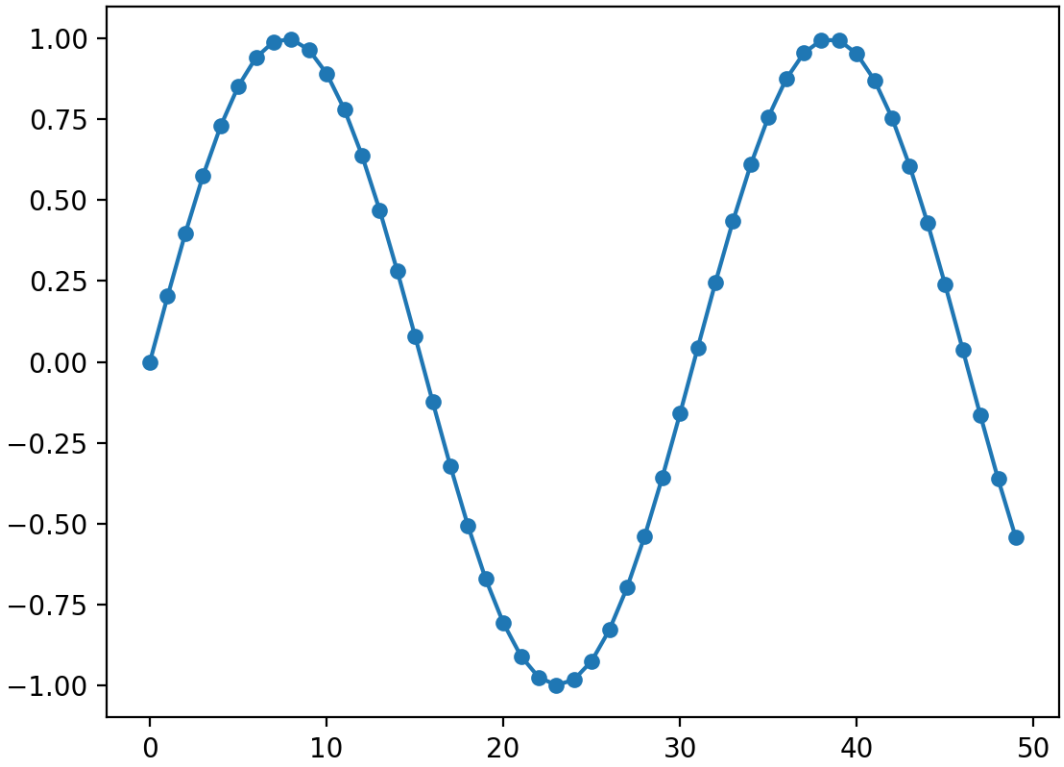
If the dataframe more than one columne, we can plot each column on separate axes

```
_ = plot(x, '-o', share_axes=False)
```



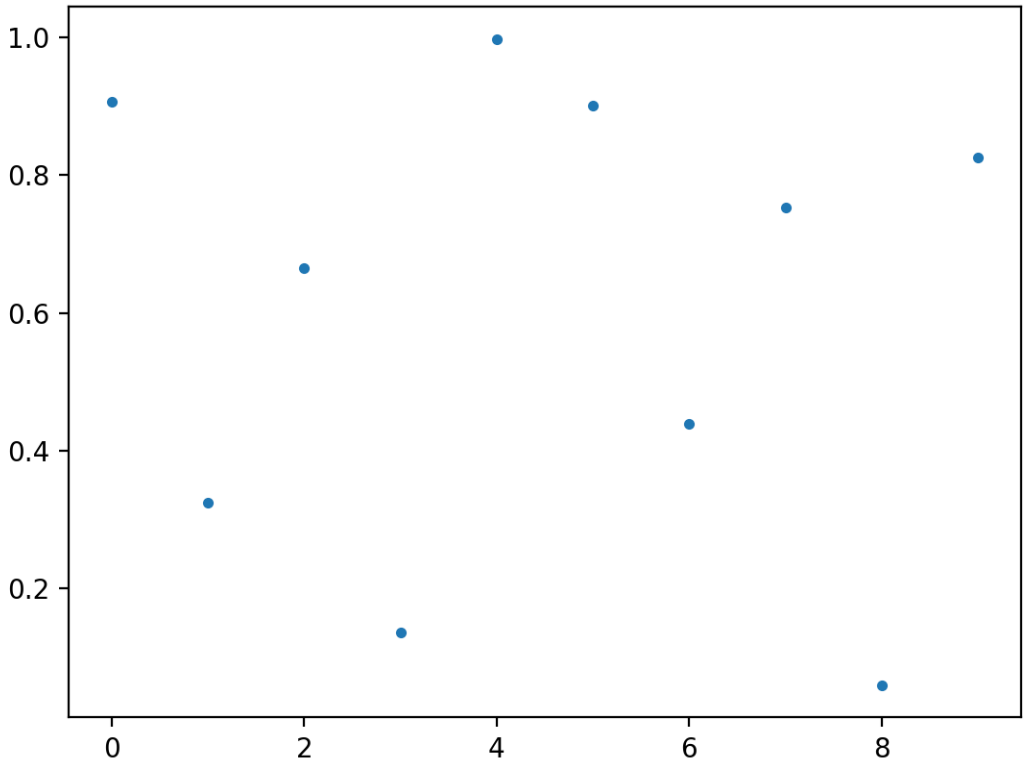
The marker size can be set using `markersize` or `ms` argument.

```
_ = plot(y, marker=".", markersize=10)
```



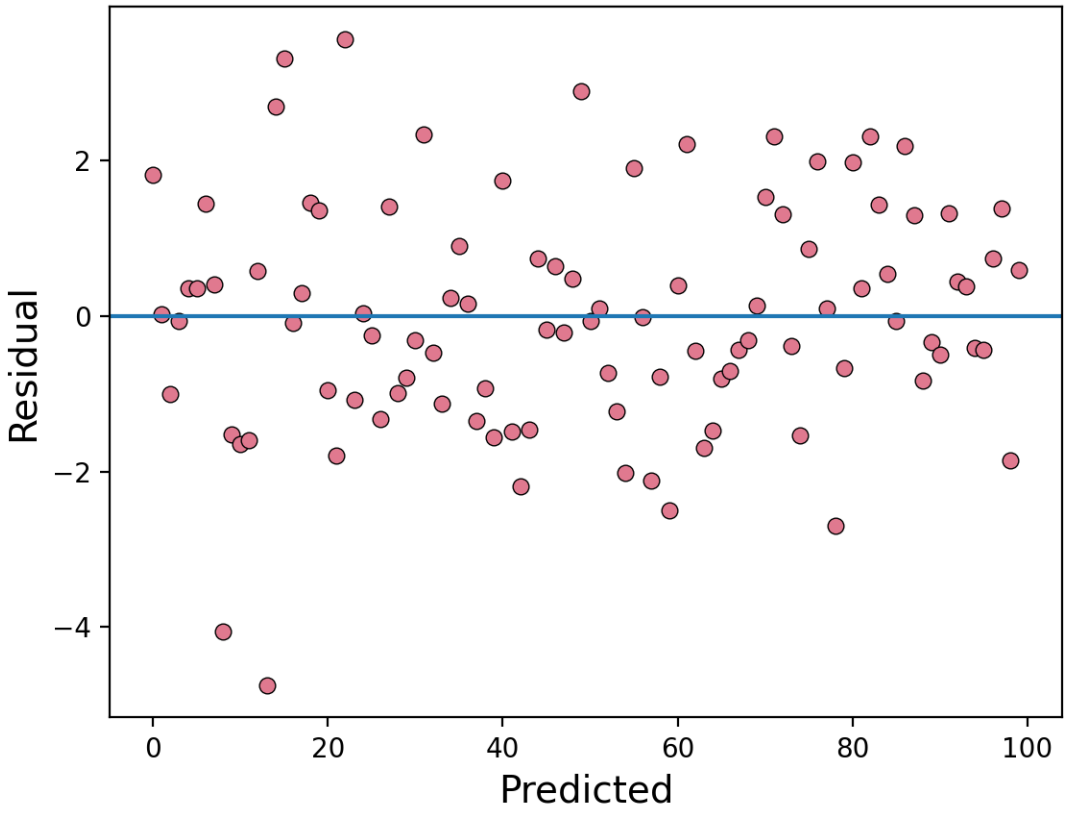
If the array contains nans, they are simply notplotted

```
x = np.append(np.random.random(10), np.nan)
_ = plot(x, '.')
```



The `plot` function returns matplotlib Axes object, which can be used for further processing.

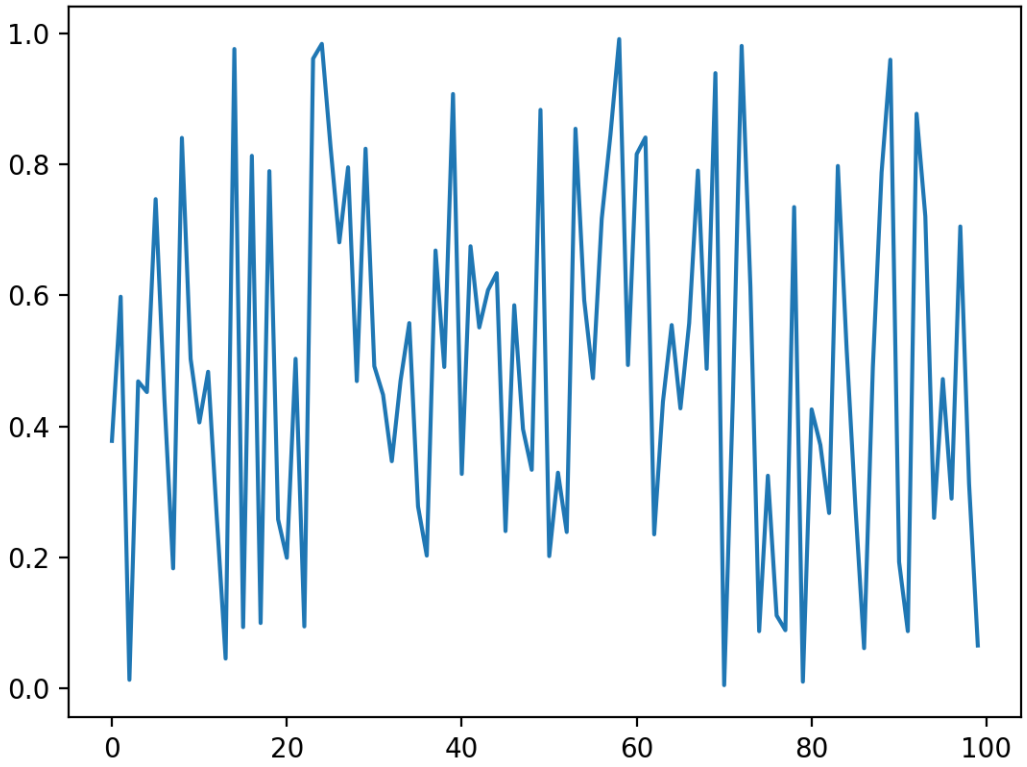
```
x = np.random.normal(size=100)
y = np.random.normal(size=100)
e = x-y
ax = plot(
    e,
    'o',
    show=False,
    markerfacecolor=np.array([225, 121, 144])/256.0,
    markeredgecolor="black", markeredgewidth=0.5,
    ax_kws=dict(
    xlabel="Predicted",
    ylabel="Residual",
    xlabel_kws={"fontsize": 14},
    ylabel_kws={"fontsize": 14}),
     )

print(f"Type of ax is: {type(ax)}")

# draw horizontal line on y=0
ax.axhline(0.0)
plt.show()
```




```
Type of ax is: <class 'matplotlib.axes._axes.Axes'>
```

We can also provide an already existing axes to `plot` function using `ax` argument.

```
_, ax = plt.subplots()
_ = plot(np.random.random(100), ax=ax)
```



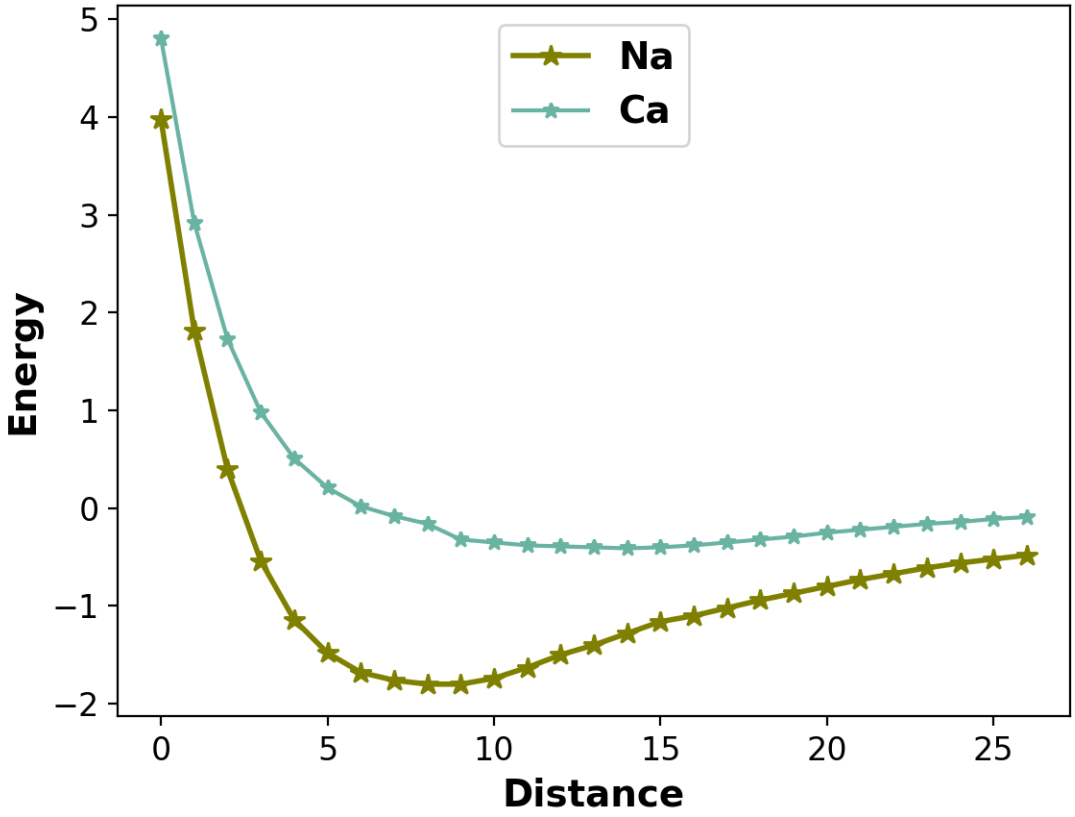
The arguments for design/manipulation of x/y axis labels and tick labels are handled by process_axis function. All the arugments of process_axis function can be given to the `plot` function.

```
y1 = [3.983,1.82,0.397,-0.54,-1.14,-1.48,-1.68,
      -1.76,-1.80,-1.80,-1.74,-1.63,-1.50,-1.40,
      -1.28,-1.16,-1.10,-1.02,-0.94,-0.87,-0.80,
      -0.73,-0.67,-0.61,-0.56,-0.52,-0.48]

y2 = [4.81, 2.92, 1.73, 0.98, 0.51, 0.21, 0.02,
      -0.08, -0.16, -0.32, -0.35, -0.38, -0.39,
      -0.40, -0.41, -0.40, -0.38, -0.35, -0.32,
      -0.29, -0.25, -0.22, -0.19, -0.16, -0.14,
      -0.11, -0.09]


plot(y1, '-*', lw=2.0, ms=8, label="Na", color="olive",  show=False)

_ = plot(y2, '-*', label="Ca", color="#69b3a2",
     ax_kws=dict(
     legend_kws = {"loc": "upper center", 'prop':{"weight": "bold", 'size': 14}},
     xlabel="Distance", xlabel_kws={"fontsize": 14, 'weight': "bold"},
     ylabel="Energy", ylabel_kws={"fontsize": 14, 'weight': 'bold'},
     xtick_kws = {'labelsize': 12},
     ytick_kws = {'labelsize': 12}),
```

(continues on next page)

(continued from previous page)

```
    )
```



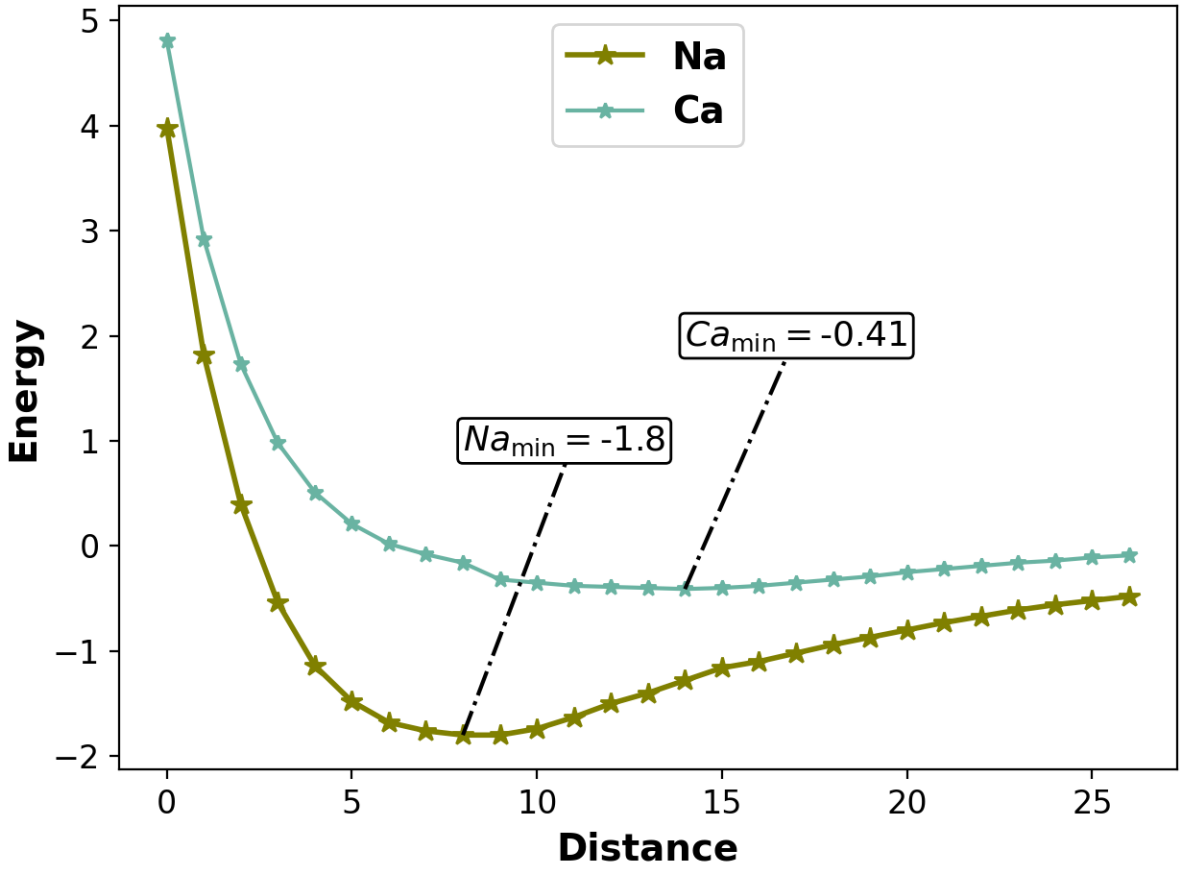
We can add text to a plot using the axes object returned by the `plot` function.

```
plot(y1, '-*', lw=2.0, ms=8, label="Na", color="olive", show=False)

ax = plot(y2, '-*', label="Ca", show=False, color="#69b3a2",
          ax_kws=dict(
     legend_kws = {"loc": "upper center", 'prop':{"weight": "bold", 'size': 14}},
     xlabel="Distance", xlabel_kws={"fontsize": 14, 'weight': "bold"},
     ylabel="Energy", ylabel_kws={"fontsize": 14, 'weight': 'bold'},
     xtick_kws = {'labelsize': 12},
     ytick_kws = {'labelsize': 12}),
     )

# Add line conecting mean value and its label
ax.plot([np.argmin(y1), 11], [np.min(y1), 1], ls="dashdot", color="black", zorder=3)

# Add mean value label.
ax.text(np.argmin(y1), 1,
    r"$Na_{\rm{min}} = $" + str(round(np.min(y1), 2)),
    fontsize=13, va="center",
    bbox=dict(facecolor="white", edgecolor="black", boxstyle="round", pad=0.15),
```

(continues on next page)

(continued from previous page)

```
    zorder=10  # to make sure the line is on top
)

# Add line conecting mean value and its label
ax.plot([np.argmin(y2), 17], [np.min(y2), 2], ls="dashdot", color="black", zorder=3)

# Add mean value label.
ax.text(np.argmin(y2), 2,
    r"$Ca_{\rm{min}} = $" + str(round(np.min(y2), 2)),
    fontsize=13, va="center",
    bbox=dict(facecolor="white", edgecolor="black", boxstyle="round", pad=0.15),
    zorder=10  # to make sure the line is on top
)

plt.tight_layout()
plt.show()
```



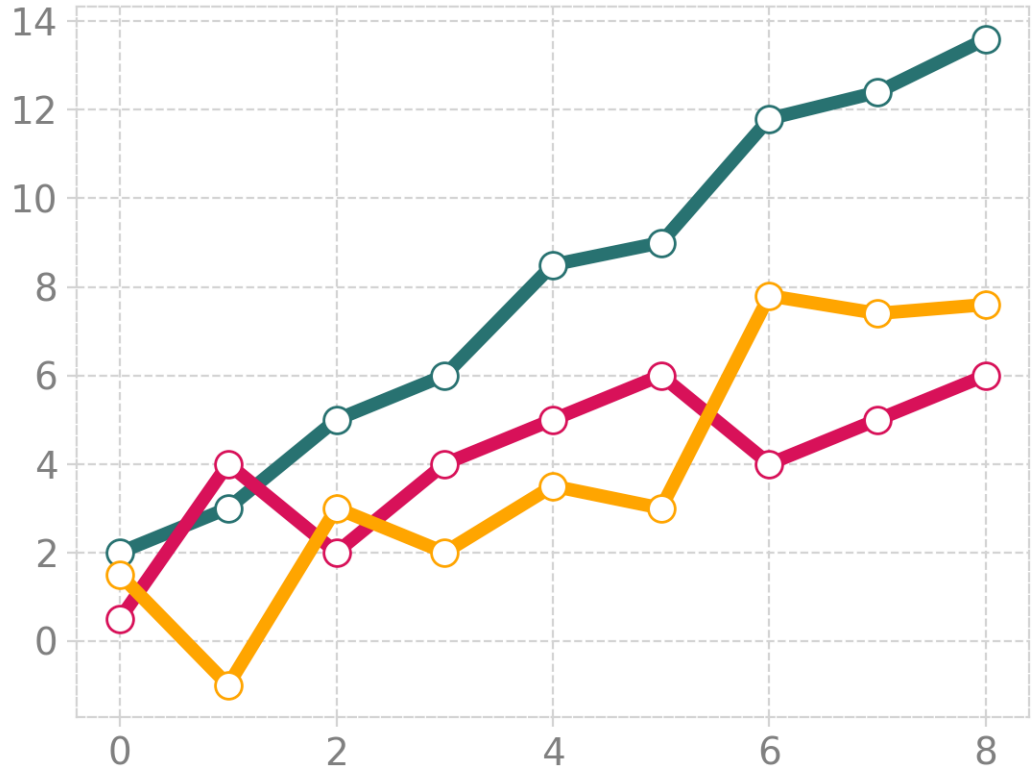
setting spine colors

```
y1 = [2, 3,5,6, 8.5, 9, 11.8, 12.4, 13.6]
y2 = [0.5, 4, 2, 4, 5, 6, 4, 5, 6]
y3 = np.array(y1) - np.array(y2)
```

(continues on next page)

(continued from previous page)

```
plot(y1, marker='o', mfc='white', ms=10, lw=5,
     color='#287271', show=False)
plot(y2, marker='o', mfc='white', ms=10, lw=5,
     color='#D81159', show=False)
ax = plot(y3, marker='o', mfc='white', ms=10, lw=5,
     color='orange', show=False)
ax.grid(ls='--', color='lightgrey')
for spine in ax.spines.values():
    spine.set_edgecolor('lightgrey')
    spine.set_linestyle('dashed')
ax.tick_params(color='lightgrey', labelsize=14, labelcolor='grey')
plt.show()
```



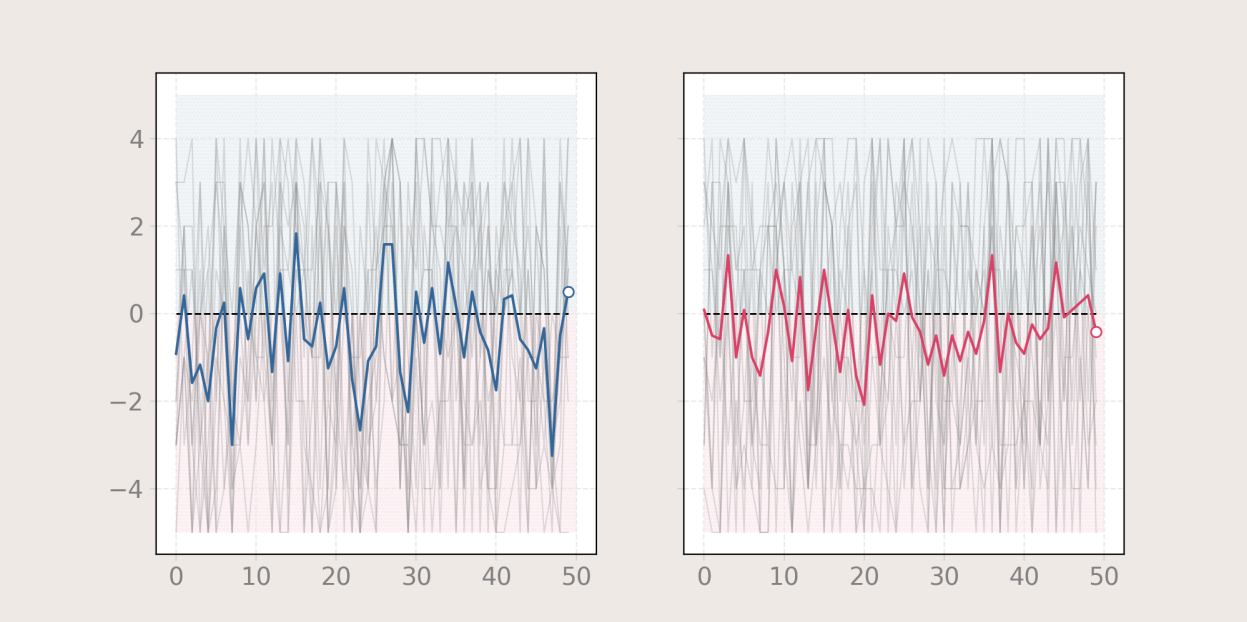
using fill between

```
n = 12
x1 = np.random.randint(-5, 5, (50, n))
x2 = np.random.randint(-5, 5, (50, n))

f, axes = plt.subplots(1, 2, figsize=(10, 5), sharey="all", facecolor = "#EFE9E6")
axes[0].grid(ls='--', color='#efe9e6', zorder=2)
axes[1].grid(ls='--', color='#efe9e6', zorder=2)
for i in range(n):
```

(continues on next page)

(continued from previous page)

```
    plot(x1[:, i], ax=axes[0], lw = .75, color = 'grey', alpha = 0.25,
         show=False)
    plot(x2[:, i], ax=axes[1], lw=.75, color='grey', alpha=0.25,
         show=False)

plot(np.zeros(50), ax=axes[0], show=False, color='black', ls='dashed', lw=1)
plot(np.zeros(50), ax=axes[1], show=False, color='black', ls='dashed', lw=1)

plot(x1.mean(axis=1), ax=axes[0], show=False,
     lw=1.5, color='#336699', zorder=5, markevery=[-1], marker='o', ms=6, mfc='white')
plot(x2.mean(axis=1), ax=axes[1], show=False,
     lw=1.5, color='#DA4167', zorder=5, markevery=[-1], marker='o', ms=6, mfc='white')

axes[0].fill_between(x=[0, 50], y1=0, y2=5, color='#336699', alpha=0.05,
                ec='None', hatch='......', zorder=1)
axes[0].fill_between(x=[0, 50], y1=0, y2=-5, color='#DA4167',
                     alpha=0.05, ec='None', hatch='......', zorder=1)

axes[0].tick_params(color='lightgrey', labelsize=14, labelcolor='grey')

axes[1].fill_between(x=[0, 50], y1=0, y2=5, color='#336699', alpha=0.05,
                ec='None', hatch='......', zorder=1)
axes[1].fill_between(x=[0, 50], y1=0, y2=-5, color='#DA4167',
                     alpha=0.05, ec='None', hatch='......', zorder=1)
axes[1].tick_params(color='lightgrey', labelsize=14, labelcolor='grey')
plt.show()
```



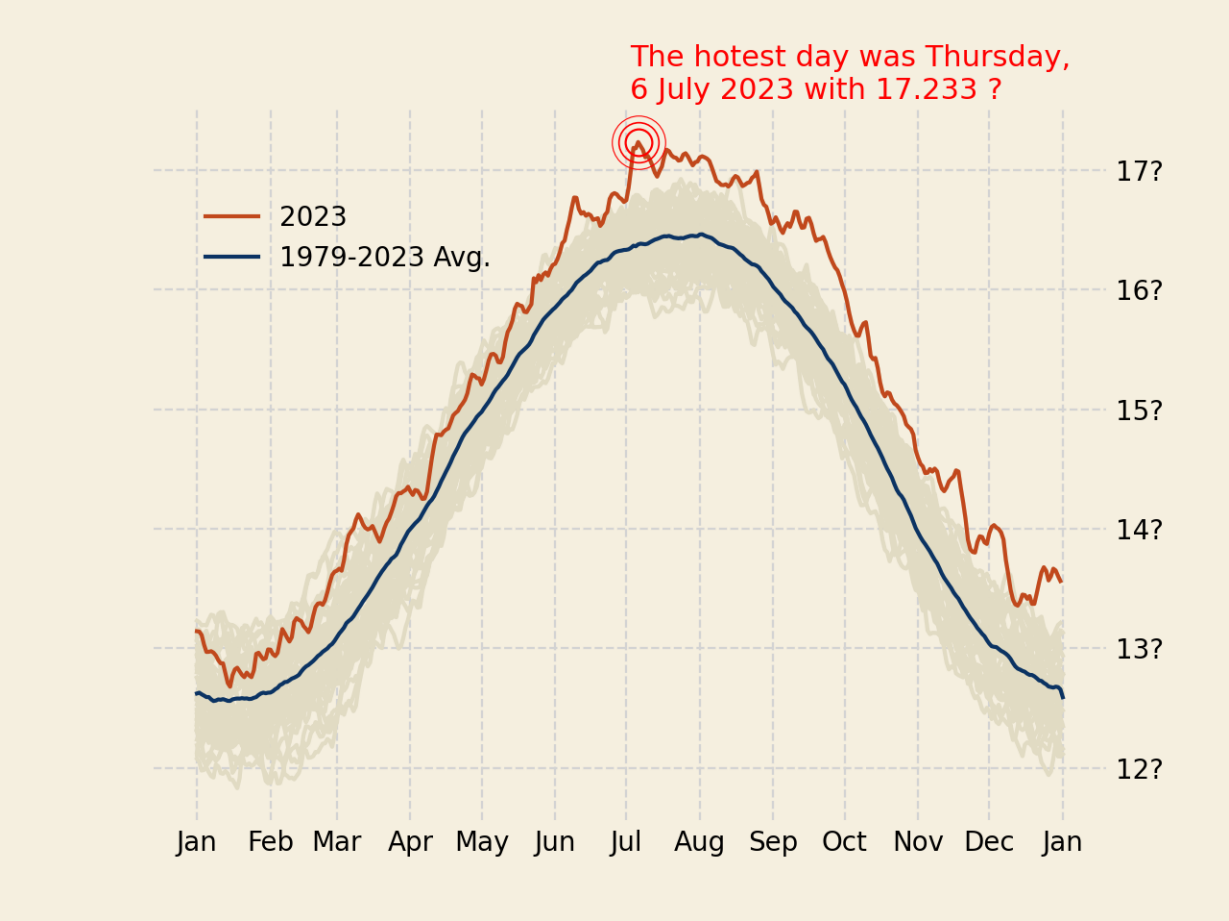
working with axes ticks and ticklabels

```
data = pd.read_json('https://climatereanalyzer.org/clim/t2_daily/json_cfsr/cfsr_world_t2_
↪day.json')
```

(continues on next page)

(continued from previous page)

```
index = data.pop('name')
data = pd.DataFrame(
    np.array([np.array(data.iloc[row, :].values[0]) for row in range(45)]),
    index=pd.to_datetime(index[0:45])
)
data = data.astype(float)

f, ax = plt.subplots(facecolor="#f5efdf",)
for i in range(len(data)):
    plot(data.iloc[i, :].values, show=False, ax=ax, color='#e1dbc3')

plot(data.iloc[-1, :],
     show=False, ax=ax, color='#c1481c', label='2023')
plot(data.mean(axis=0), ax=ax, color='#0b3363', show=False,
     label="1979-2023 Avg.")

yticklabels = []
for label in ax.get_yticklabels():
    yticklabels.append(f"{label.get_text()}?")
ax.set_yticklabels(yticklabels)

ax.tick_params(axis=u'both', which=u'both',length=0) # Hide ticks but show tick labels
ax.yaxis.tick_right()
# show month names as tick labels
ax.xaxis.set_major_locator(mdates.MonthLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b'))
# Remove y label
ax.set_ylabel('')
ax.legend(frameon=False, fancybox=False, bbox_to_anchor=(0.38, 0.9))

# setting grid, facecolor and spines
ax.grid(visible=True, ls='--', color='lightgrey')
ax.set_facecolor('#f5efdf')
despine_axes(ax)

ts = pd.concat([data.iloc[i, :] for i in range(data.shape[0])]).dropna()
ts.index = pd.date_range(data.index[0], periods=len(ts), freq="D")
max_temp = ts.idxmax()
ax.text(0.5, 1.05,
    f"""The hotest day was {max_temp.day_name()},
{max_temp.day} {max_temp.month_name()} {max_temp.year} with {data.max().max()} ?""",
    fontsize=11, va="center",
        color="red", zorder=10,
        transform=ax.transAxes
)
ax.plot(data.iloc[-1, :].idxmax(), data.iloc[-1, :].max(), 'ro',
        markersize=10, fillstyle='none', markeredgewidth=0.8)
ax.plot(data.iloc[-1, :].idxmax(), data.iloc[-1, :].max(), 'ro',
        markersize=15, fillstyle='none', markeredgewidth=0.6)
ax.plot(data.iloc[-1, :].idxmax(), data.iloc[-1, :].max(), 'ro',
        markersize=20, fillstyle='none', markeredgewidth=0.4)
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_plot.py:323: UserWarning: set_ticklabels() should only be used with a fixed␣
↪number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_yticklabels(yticklabels)
```

Total running time of the script: (0 minutes 8.674 seconds)

6.2 scatter plot

```
# sphinx_gallery_thumbnail_number = 4

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from easy_mpl.utils import add_cbar
from matplotlib.lines import Line2D
from easy_mpl.utils import version_info
from easy_mpl.utils import map_array_to_cmap

from easy_mpl import scatter
```

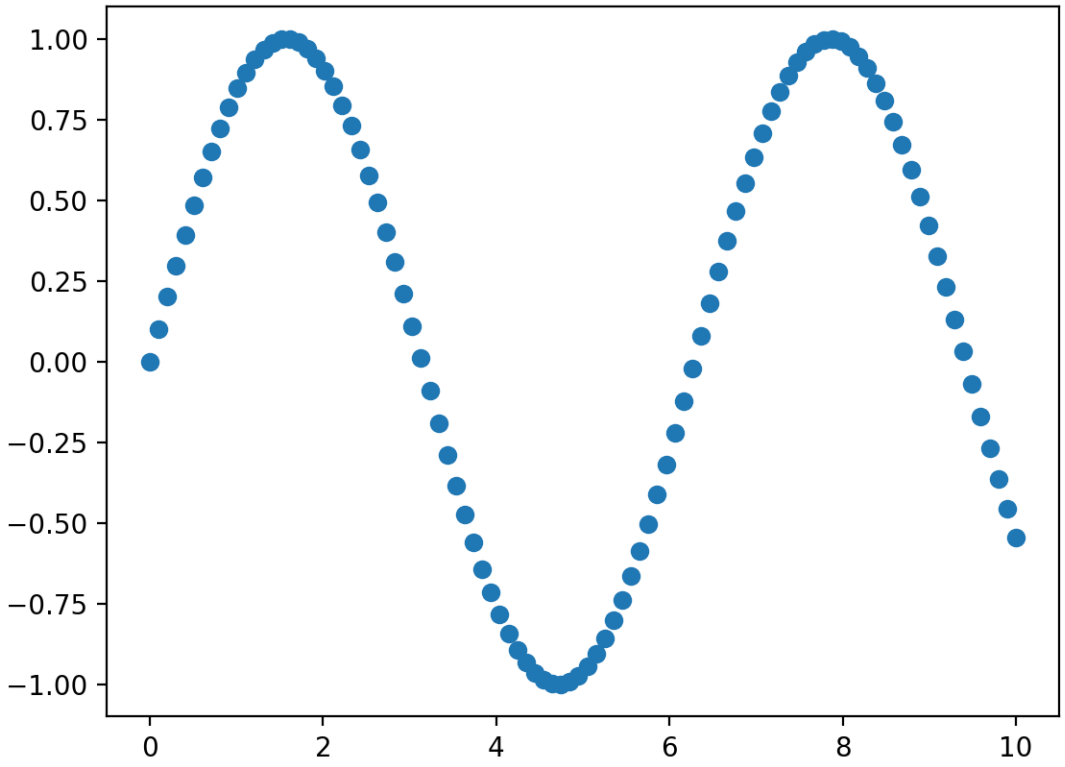
(continues on next page)

(continued from previous page)

```
f = "https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/datasets/arg_
↪busan.csv"
dataframe = pd.read_csv(f, index_col='index')
dataframe = dataframe[['tide_cm', 'pcp_mm', 'sal_psu', 'pcp12_mm',
                       'sul1_coppml', 'tetx_coppml', 'blaTEM_coppml', 'aac_coppml']]

version_info()  # print version information of all the packages being used
```

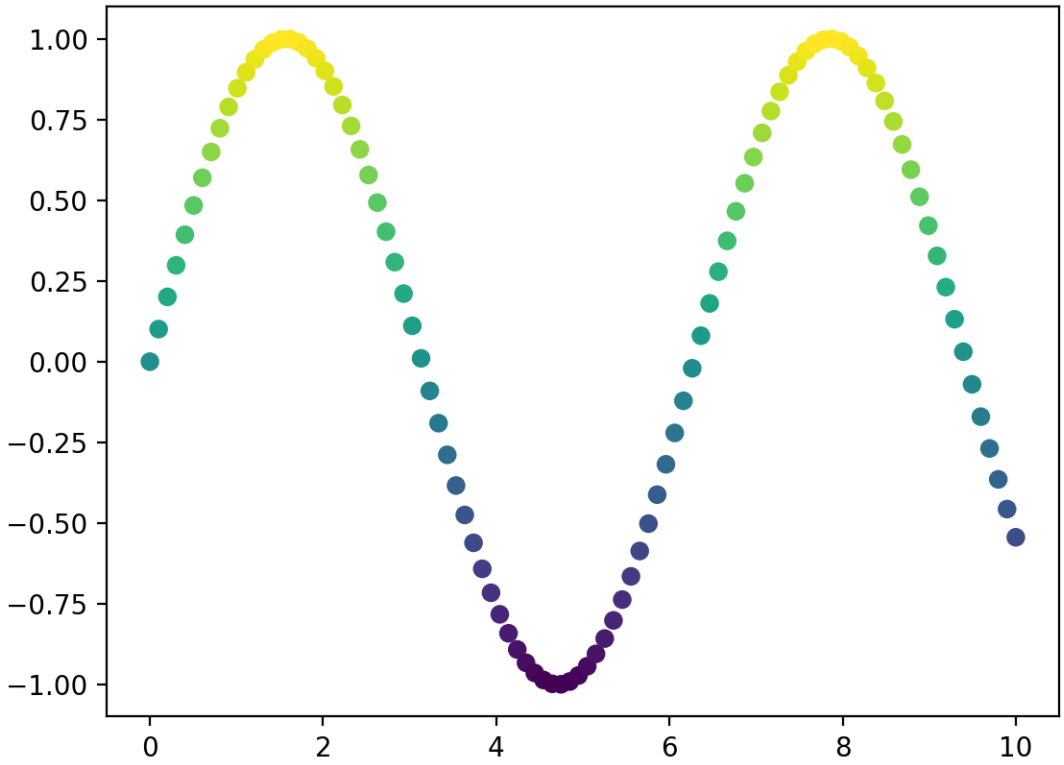
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

_ = scatter(x, y)
```



Instead of drawing all the markers in same color, we can make the color to show something useful. Below, the color represents y values.

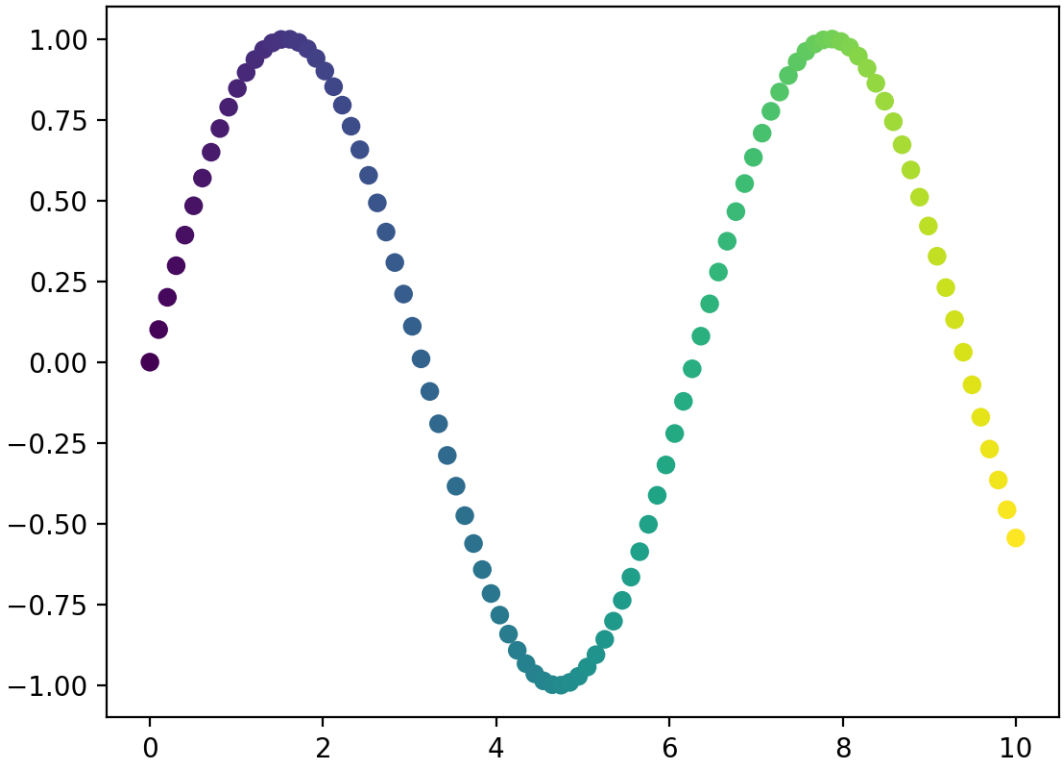
```
_ = scatter(x, y, c=y)
```



As the value of `y` goes higher, the color of marker becomes yellowish. On the other hand, as the value of `y` goes lower, the color becomes bluish.

Instead of making the color to show values of *y*, can use another array for the color.

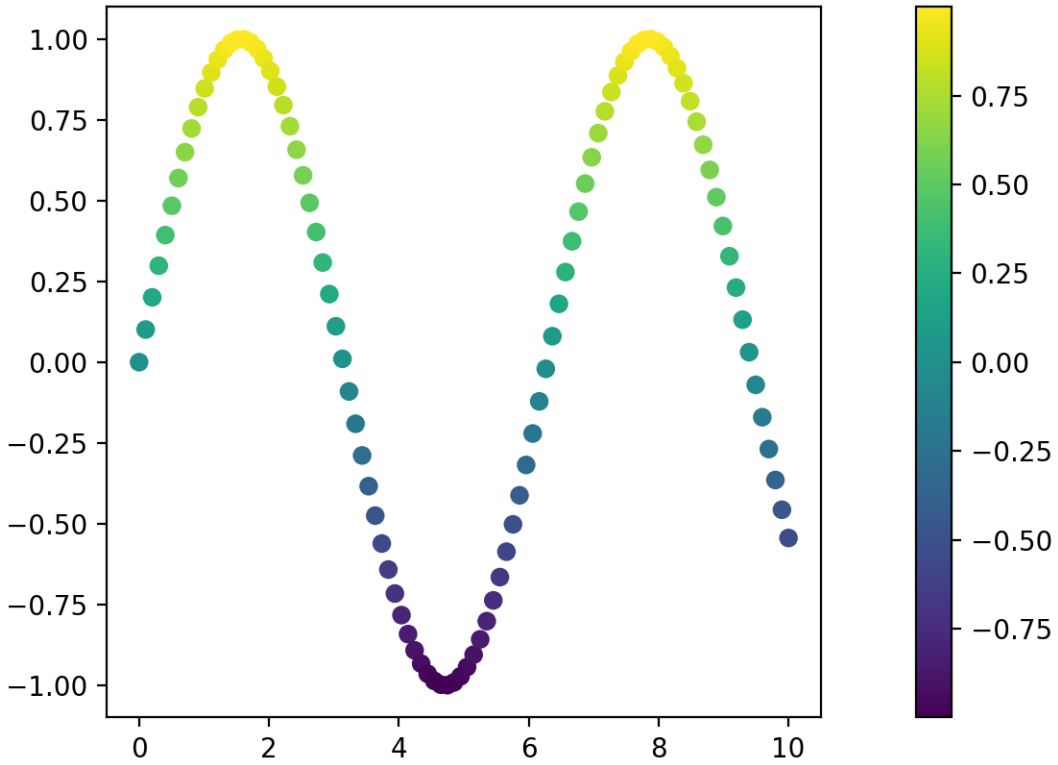
```
z = np.arange(100)
_ = scatter(x, y, c=z)
```



Now the color of marker changes from left to right instead of from bottom to top.

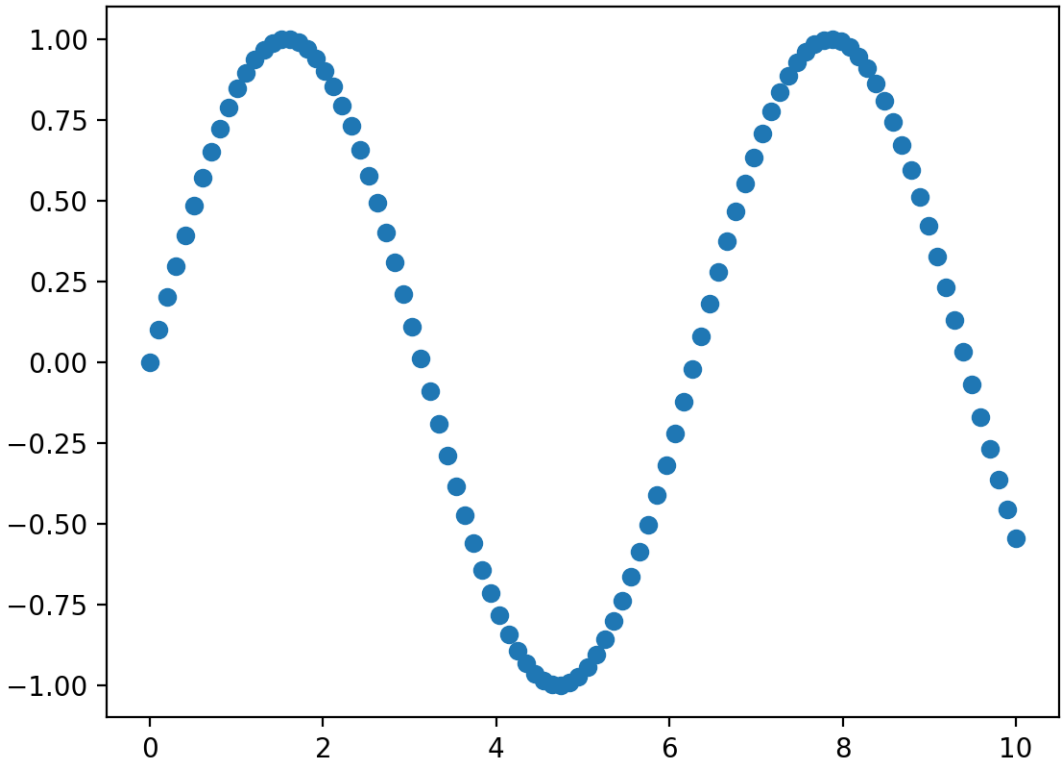
We can show the colorbar by setting the `colorbar` to True.

```
_ = scatter(x, y, c=y, colorbar=True)
```



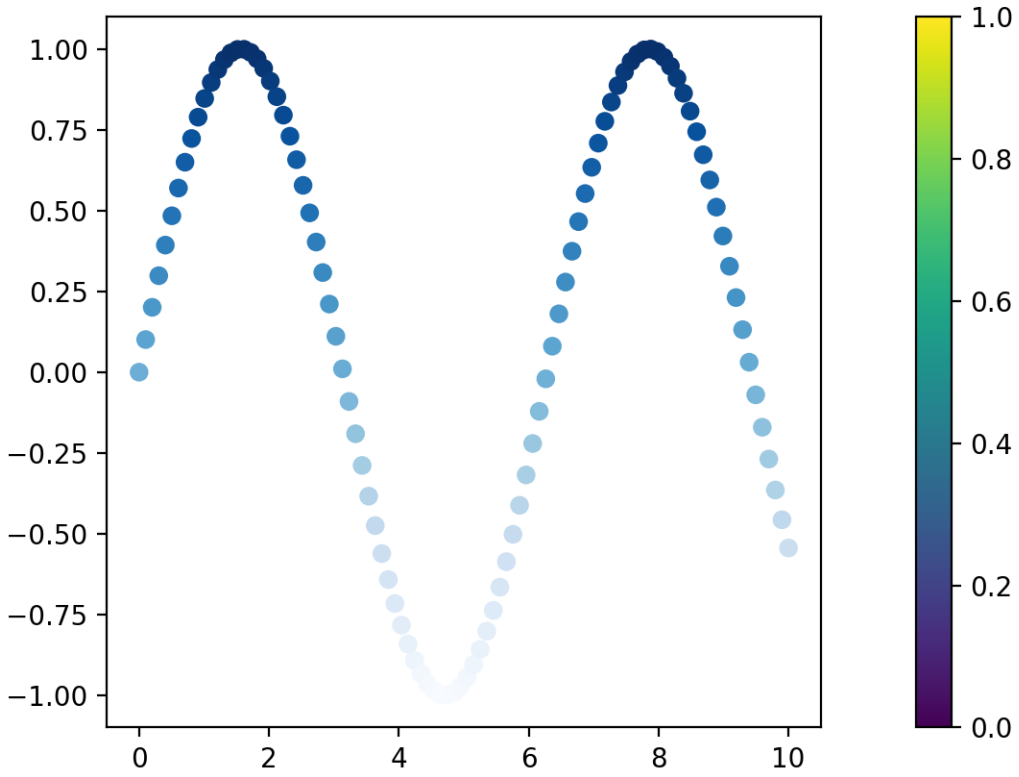
The function `scatter` returns a tuple. The first argument is a matplotlib axes which can be used for further processing

```
axes, _ = scatter(x, y, show=False)
assert isinstance(axes, plt.Axes)
```



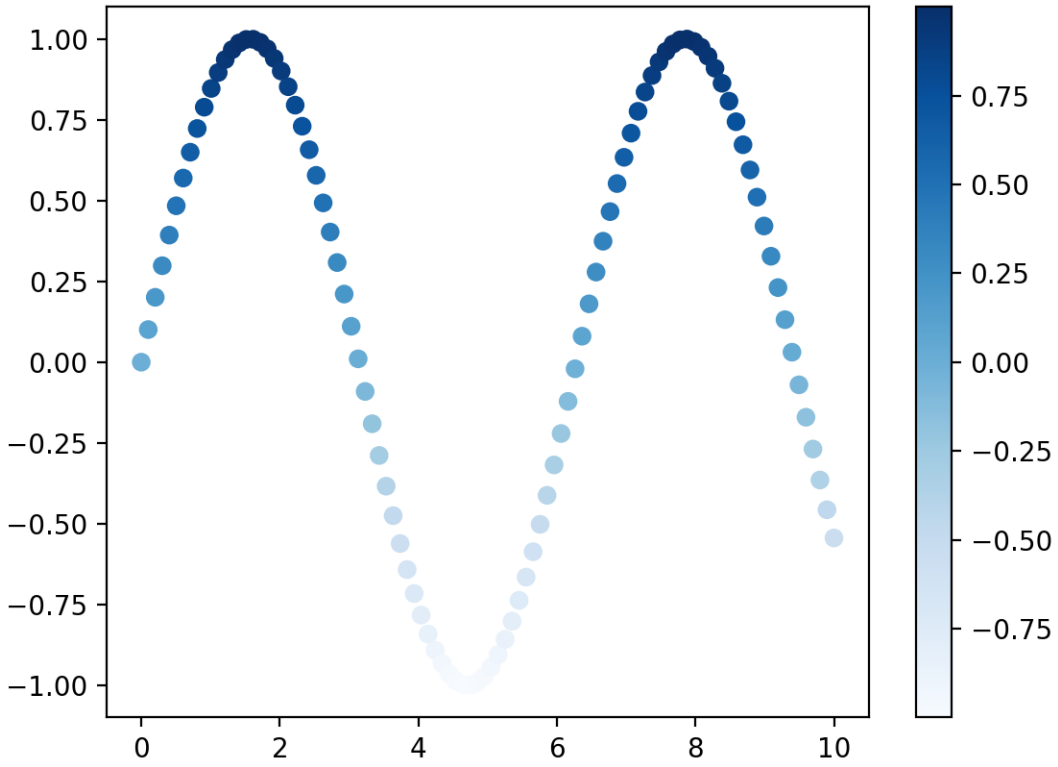
We can provide the actual values of rbg as list/array to color/c argument.

```
colors, _ = map_array_to_cmap(y, "Blues")
_ = scatter(x, y, color=colors, colorbar=True)
```



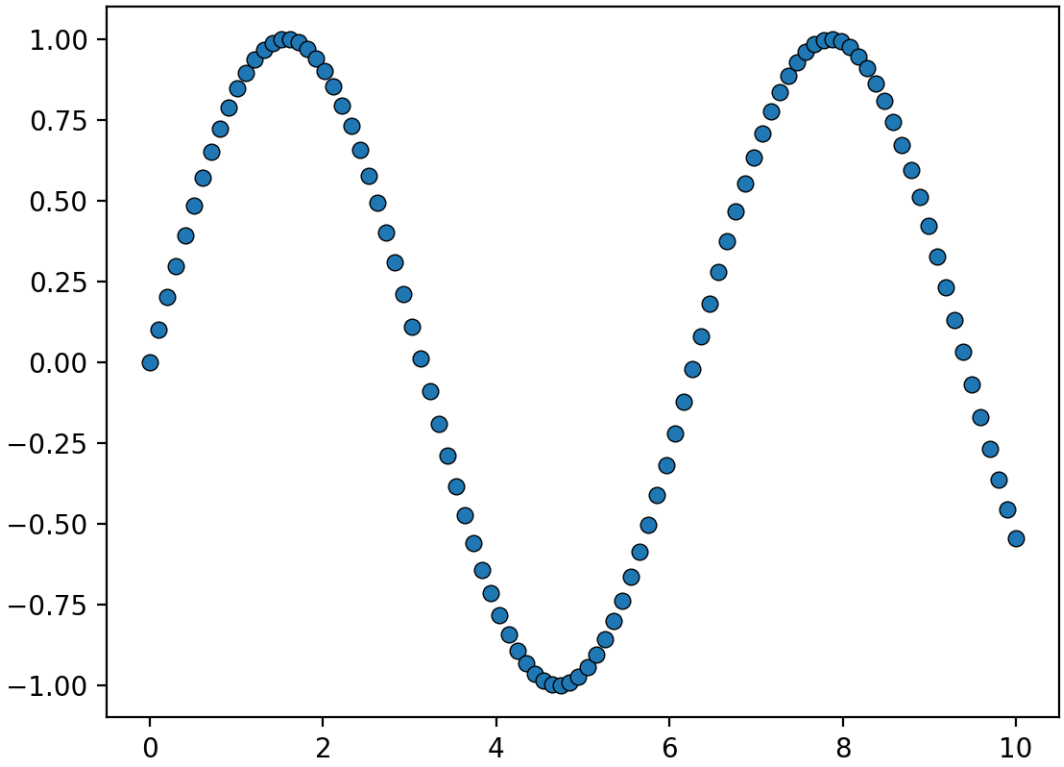
However, if we show the colorbar, the colorbar in such a case will be wrong.

```
colors, mapper = map_array_to_cmap(y, "Blues")
ax, sc = scatter(x, y, color=colors, show=False)
plt.colorbar(mapper, ax=ax)  # we must provide the mapper to ``colorbar`` otherwise␣
↪colorbar will be wrong
plt.show()
```



The properties of the markers can be manipulated.

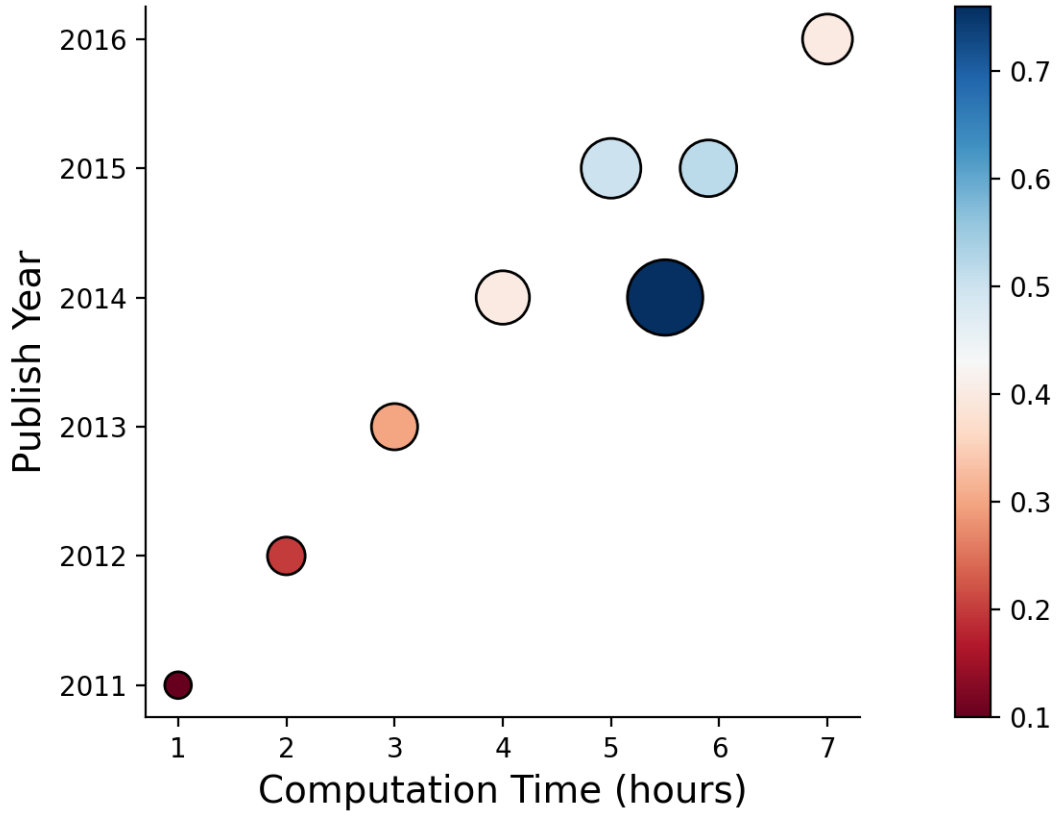
```
_ = scatter(x, y, edgecolors='black', linewidth=0.5)
```



We can use any color map to show the marker colors. A complete list of valid matplotlib colormaps can be found here

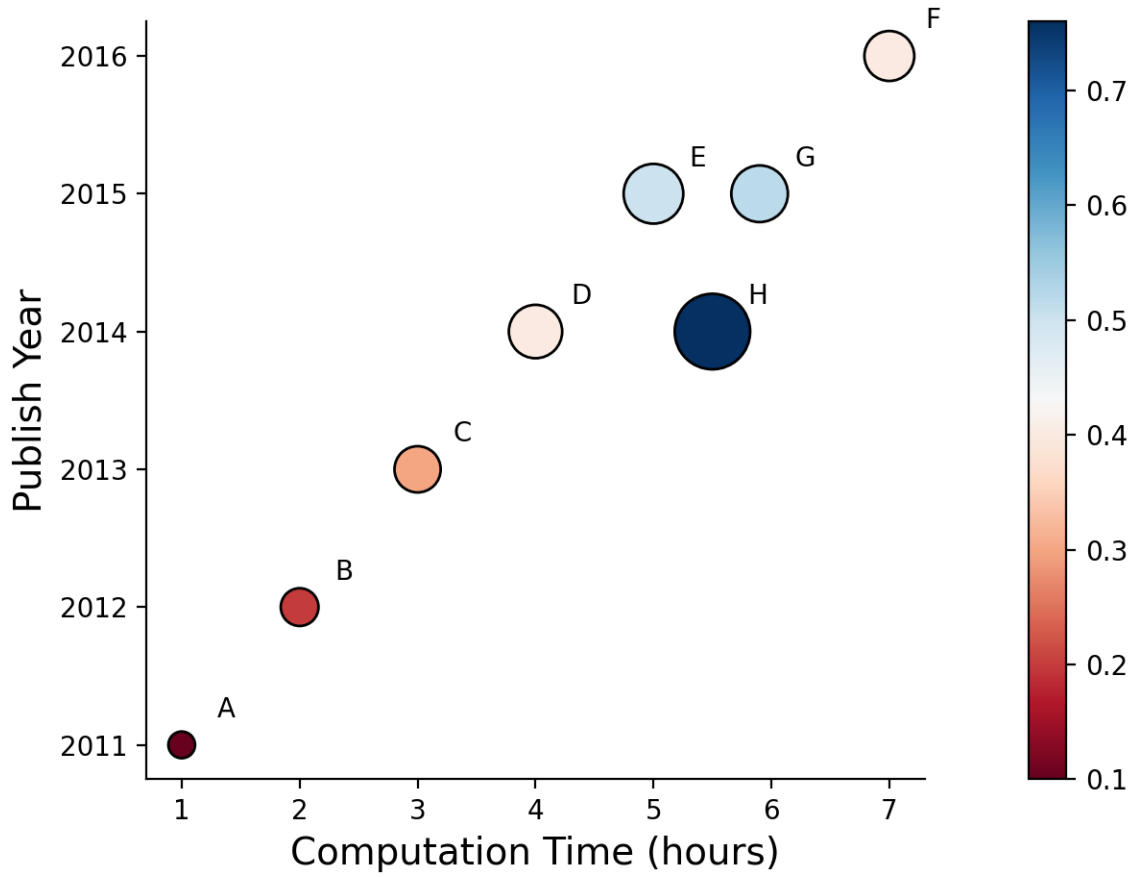
The size of the marker can be tuned using `s` keyword. If a single value is provided, all markers will be of single size. We can however make each maker of variable size by passing an array to `s` keyword.

```
time = [1, 2, 3, 4, 5, 7, 5.9, 5.5]
parameters = [100, 200, 300, 400, 500, 350, 450, 800]
performance = [0.1, 0.2, 0.3, 0.4, 0.5, 0.4, 0.52, 0.76]
year = [2011, 2012, 2013, 2014, 2015, 2016, 2015, 2014]

_ = scatter(time, year, c=performance, s=parameters,
        colorbar=True,
        edgecolors='black', linewidth=1.0,
        cmap="RdBu",
        ax_kws={"xlabel":"Computation Time (hours)", 'ylabel': "Publish Year",
                'xlabel_kws': {"fontsize": 14},'ylabel_kws': {"fontsize": 14},
                'top_spine': False, 'right_spine': False})
```



We can also annotate the markers by providing `marker_labels` argument.

```
time = [1, 2, 3, 4, 5, 7, 5.9, 5.5]
parameters = [100, 200, 300, 400, 500, 350, 450, 800]
performance = [0.1, 0.2, 0.3, 0.4, 0.5, 0.4, 0.52, 0.76]
year = [2011, 2012, 2013, 2014, 2015, 2016, 2015, 2014],
names = ["A", "B", "C", "D", "E", "F", "G", "H"]

_ = scatter(time, year, c=performance, s=parameters,
        colorbar=True,
        edgecolors='black', linewidth=1.0,
        cmap="RdBu",
        marker_labels=names,
        yoffset=0.2,
        xoffset=0.3,
        ax_kws={"xlabel":"Computation Time (hours)", 'ylabel': "Publish Year",
                'xlabel_kws': {"fontsize": 14},'ylabel_kws': {"fontsize": 14},
                'top_spine': False, 'right_spine': False, 'tight_layout': True})
```



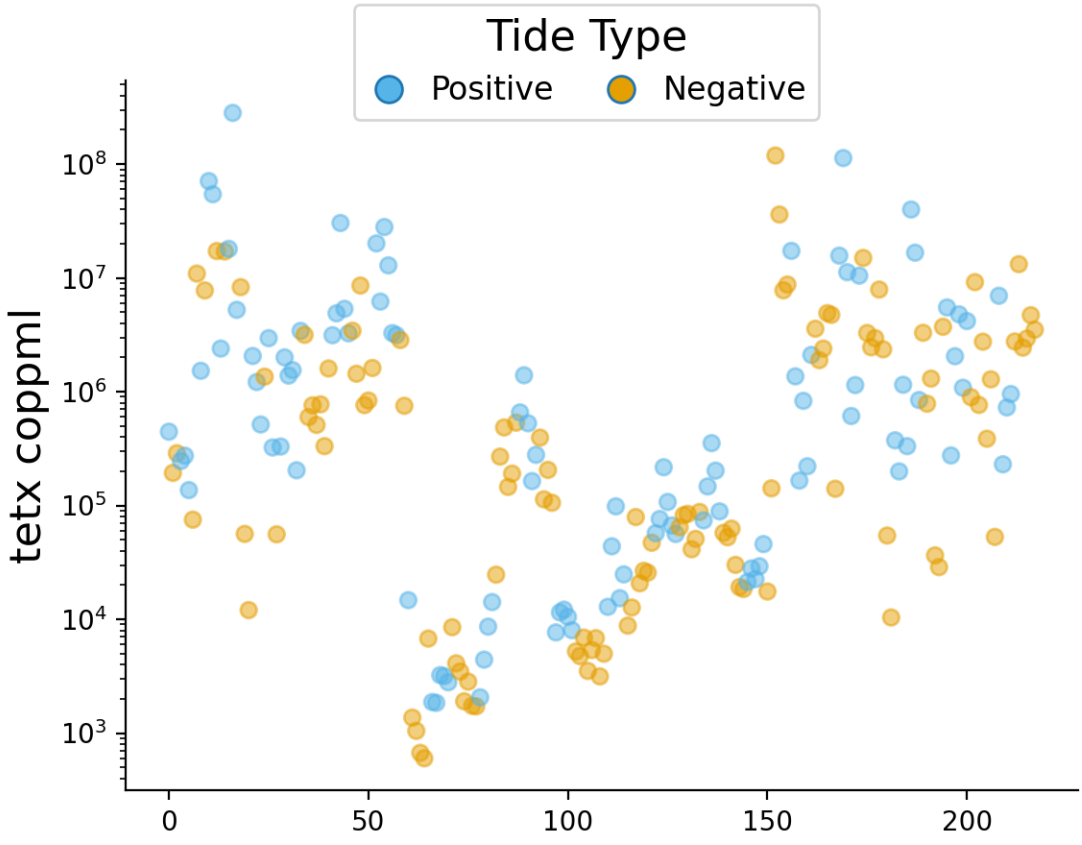
unique colors for group of values

```
df = dataframe.dropna().reset_index(drop=True)
tide = df['tide_cm']
tetx = df['tetx_coppml']
colors = np.full(len(tide), fill_value="#E69F00")
colors[np.argwhere(tide.values<0.0)] =  "#56B4E9"
ax, pc = scatter(np.arange(len(tide)), tetx,
        ax_kws=dict(logy=True, ylabel="tetx coppml", ylabel_kws={"fontsize": 16},
                    top_spine=False, right_spine=False),
        color=colors, alpha=0.5, zorder=10)
fig = ax.get_figure()
# Create handles for lines.
handles = [
    Line2D(
        [], [], label=label,
        lw=0, # there's no line added, just the marker
        marker="o", # circle marker
        markersize=10,
        markerfacecolor=colors[idx], # marker fill color
    )
    for idx, label in enumerate(['Positive', 'Negative'])
]
```

(continues on next page)

(continued from previous page)

```
# Add legend ----------------------------------------------------
legend = fig.legend(
    handles=handles,
    bbox_to_anchor=[0.5, 0.9], # Located in the top-mid of the figure.
    fontsize=12,
    handletextpad=0.6, # Space between text and marker/line
    handlelength=1.4,
    columnspacing=1.4,
    loc="center",
    ncol=6,
title_fontsize=16,
        title="Tide Type"
)
fig.show()
```



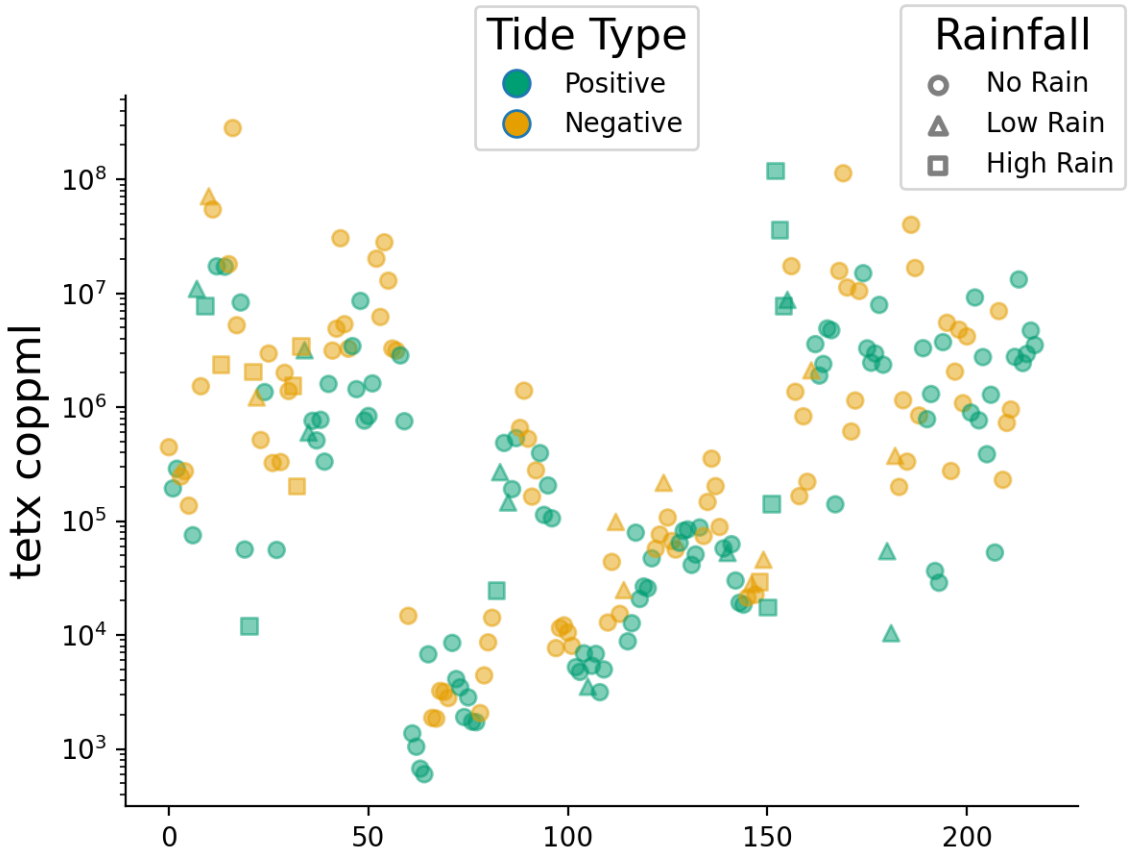
marker style for group of values

```
colors = "#009E73", "#E69F00"
def make_color(array):
    clrs = np.full(len(array), fill_value=colors[0])
    clrs[np.argwhere(array < 0.0)] = colors[1]
    return clrs
```

(continues on next page)

(continued from previous page)

```
markers = ["o", "^", "s"] # circle, triangle, square
labels = ["No Rain", "Low Rain", "High Rain"]
Y = [df.loc[df['pcp_mm']<=0.0],
        df.loc[(df['pcp_mm']>0.0) & (df['pcp_mm']<=1.0)],
        df.loc[df['pcp_mm']>1.0]]

X = [df.loc[df['pcp_mm']<=0.0].index,
        df.loc[(df['pcp_mm']>0.0) & (df['pcp_mm']<=1.0)].index,
        df.loc[df['pcp_mm']>1.0].index]

_ = ax = plt.subplots()
for label, marker, x, y in zip(labels, markers, X, Y):
    color = make_color(y['tide_cm'].values)
    axes, pc = scatter(x=x, y=y['tetx_coppml'], marker=marker,
            ax_kws=dict(logy=True, ylabel="tetx coppml", ylabel_kws={"fontsize": 16},
                                            top_spine=False, right_spine=False),
            color=color, alpha=0.5, zorder=10,
            label=label,
            show=False)

handles = [Line2D([], [], label=label,
                  marker="o", markersize=10, lw=0, markerfacecolor=colors[idx])
    for idx, label in enumerate(['Positive', 'Negative'])
]
fig = axes.get_figure()
legend = fig.legend(handles=handles, bbox_to_anchor=[0.5, 0.9],
title_fontsize=16,title="Tide Type", loc="center")

leg = plt.legend(bbox_to_anchor=[0.8, 0.85], title="Rainfall", title_fontsize=16)
for h in leg.legendHandles:
    h.set_facecolor('white')
    h.set_edgecolor('k')
    h.set_linewidth(2.0)
plt.show()
```



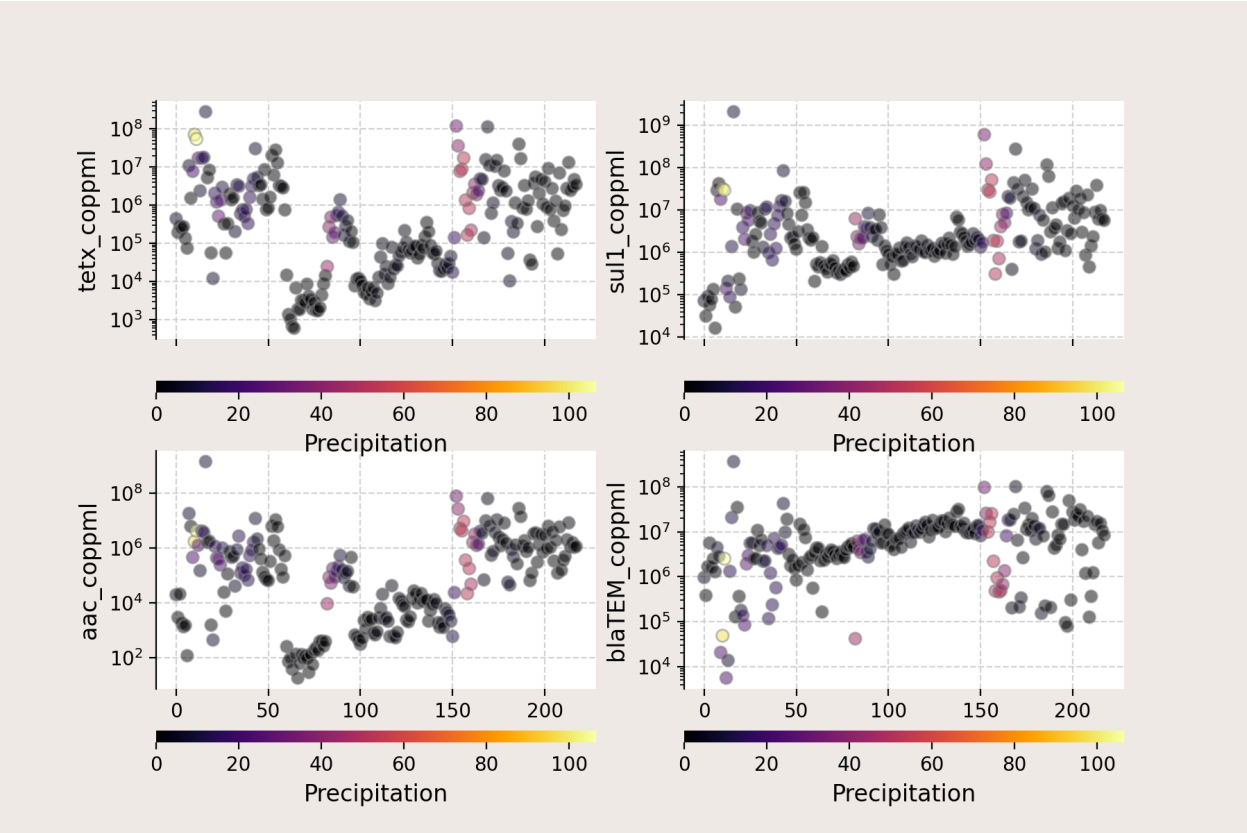

```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_scatter.py:205: MatplotlibDeprecationWarning: The legendHandles attribute␣
↪was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use␣
↪legend_handles instead.
  for h in leg.legendHandles:
```

```
df = dataframe.dropna().reset_index(drop=True)

def draw_scatter(target, ax):
    #``visible`` argument for ``ax.grid`` not available in
    # matplotlib version 3.3
    ax.grid(visible=True, ls='--', color='lightgrey')
    c, mapper = map_array_to_cmap(df['pcp12_mm'].values, "inferno")
    ax_, _ = scatter(np.arange(len(df)), df[target],
                     color=c, alpha=0.5, s=40, ec="grey", zorder=10,
                     ax_kws=dict(logy=True, ylabel=target, ylabel_kws={"fontsize": 12},
                                 top_spine=False, right_spine=False, bottom_
↪spine=False),
                     ax=ax, show=False)
    add_cbar(ax_, mappable=mapper, orientation="horizontal", pad=0.3,
                 border=False,
                 title="Precipitation", title_kws=dict(fontsize=12))
    return
```

(continues on next page)

(continued from previous page)

```
f, all_axes = plt.subplots(2,2, sharex="all", facecolor="#EFE9E6", figsize=(9, 6))
targets = ["tetx_coppml", "sul1_coppml", "aac_coppml", "blaTEM_coppml"]
for col, axes in zip(targets, all_axes.flatten()):
    draw_scatter(col, axes)
plt.show()
```



Total running time of the script: (0 minutes 5.399 seconds)

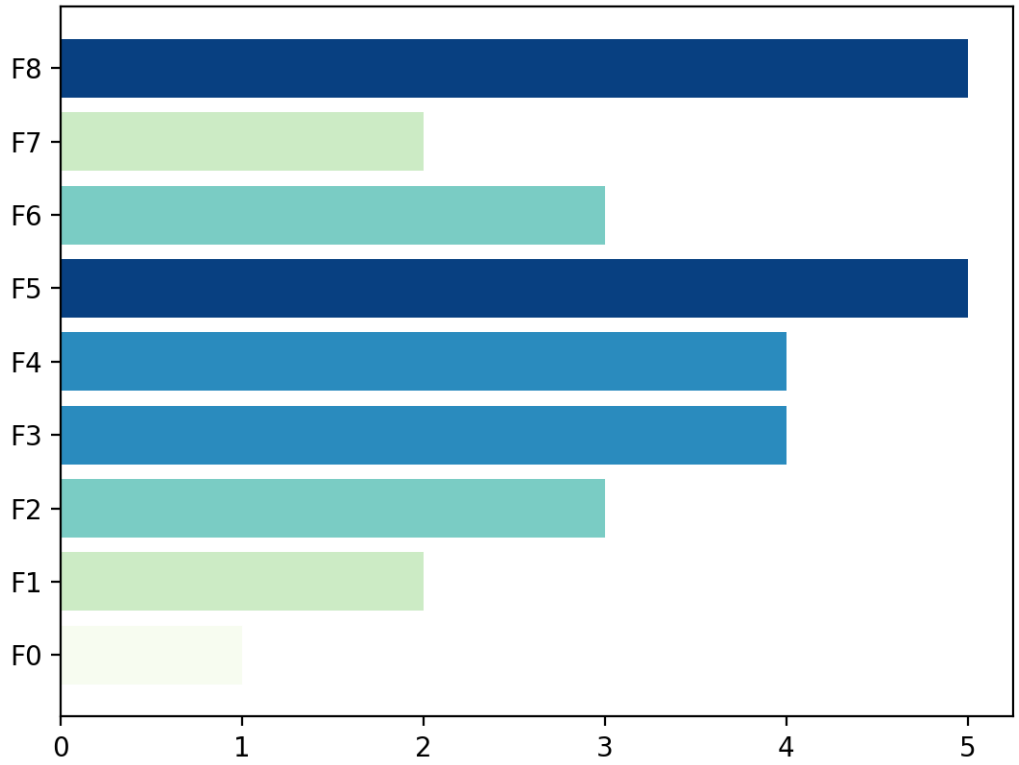
6.3 bar plot

```
# sphinx_gallery_thumbnail_number = 4

import numpy as np
from easy_mpl import bar_chart, plot
import matplotlib.pyplot as plt
from easy_mpl.utils import version_info
from easy_mpl.utils import despine_axes

version_info()  # print version information of all the packages being used
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

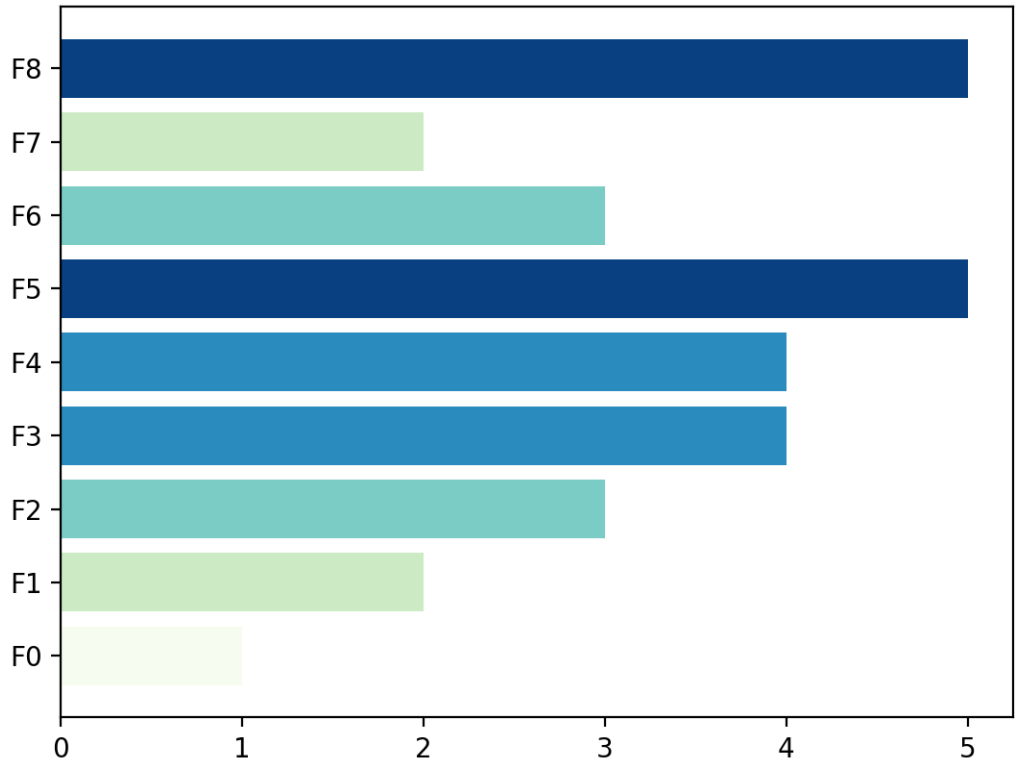
A basic chart requires just a list of values to represent as bars.

```
_ = bar_chart([1,2,3,4,4,5,3,2,5])
```



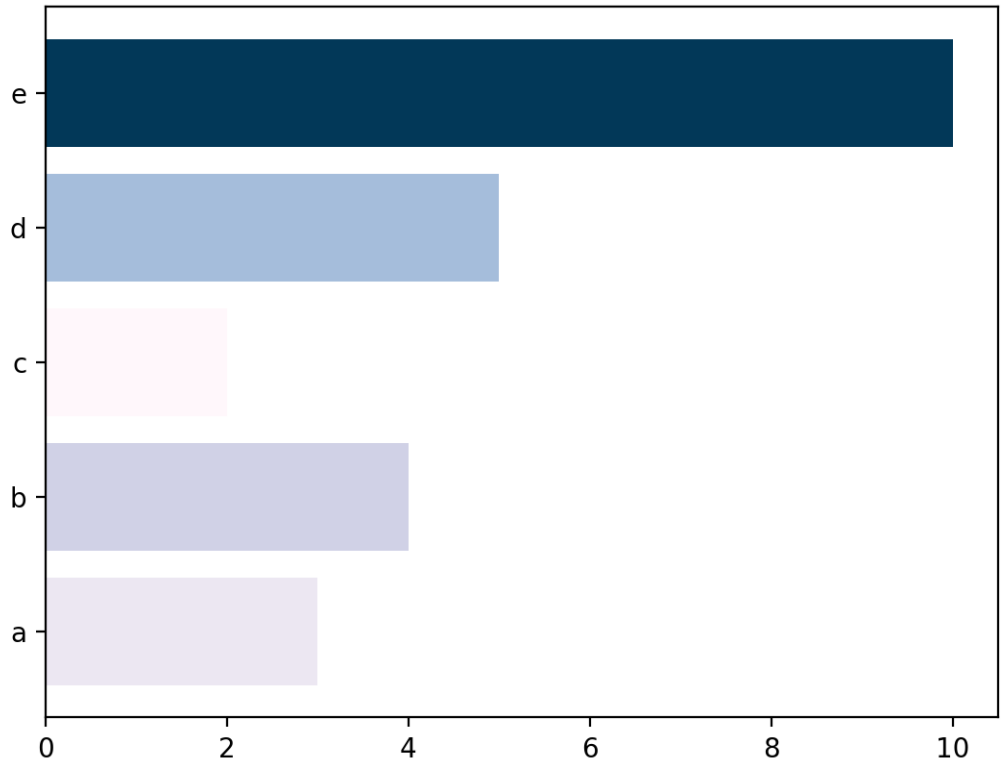
we can also provide a numpy array instead

```
_ = bar_chart(np.array([1,2,3,4,4,5,3,2,5]))
```



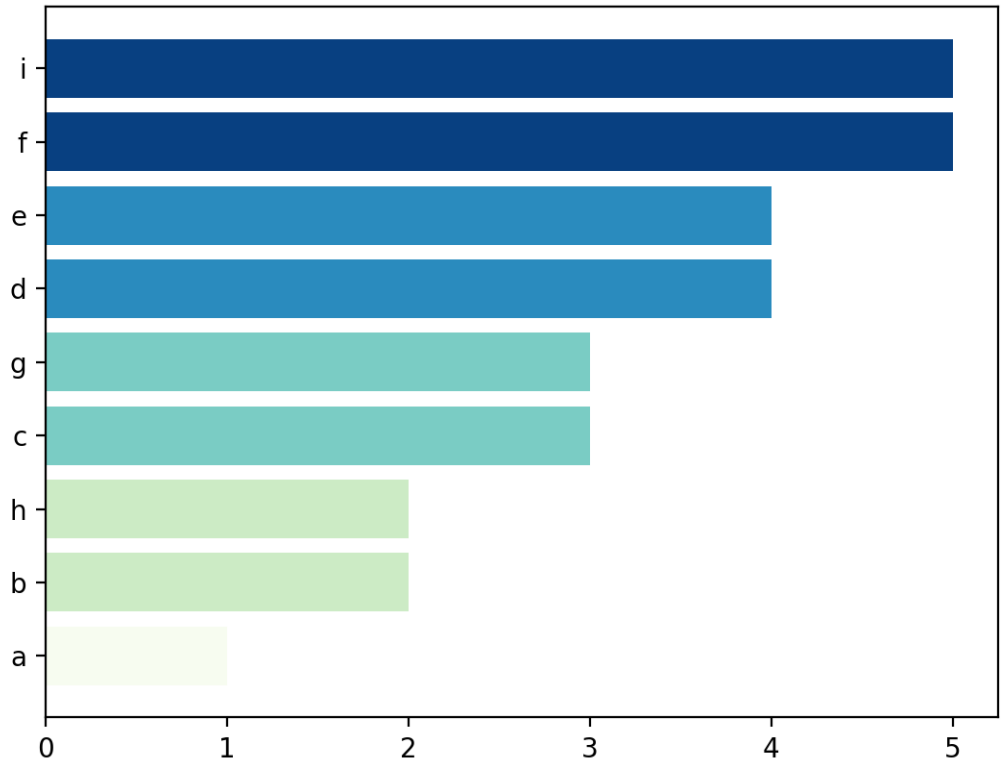
setting the labels for axis

```
_ = bar_chart([3,4,2,5,10], ['a', 'b', 'c', 'd', 'e'])
```



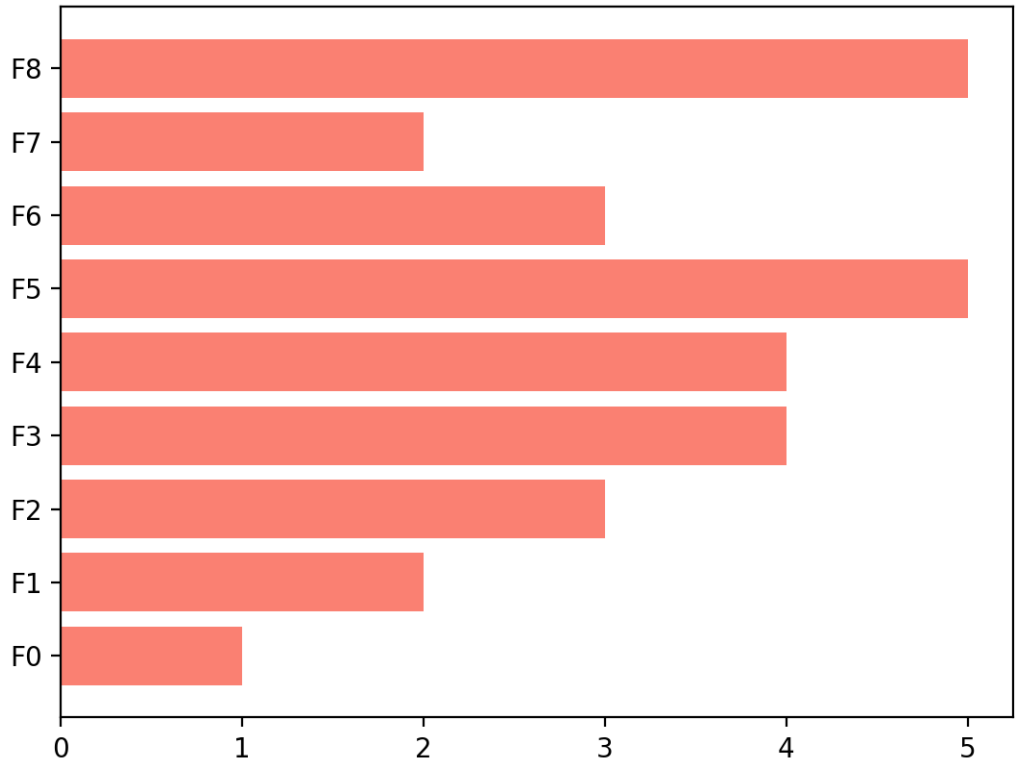
sorting the bars according to their values

```
_ = bar_chart([1,2,3,4,4,5,3,2,5],
    ['a','b','c','d','e','f','g','h','i'],
          sort=True)
```



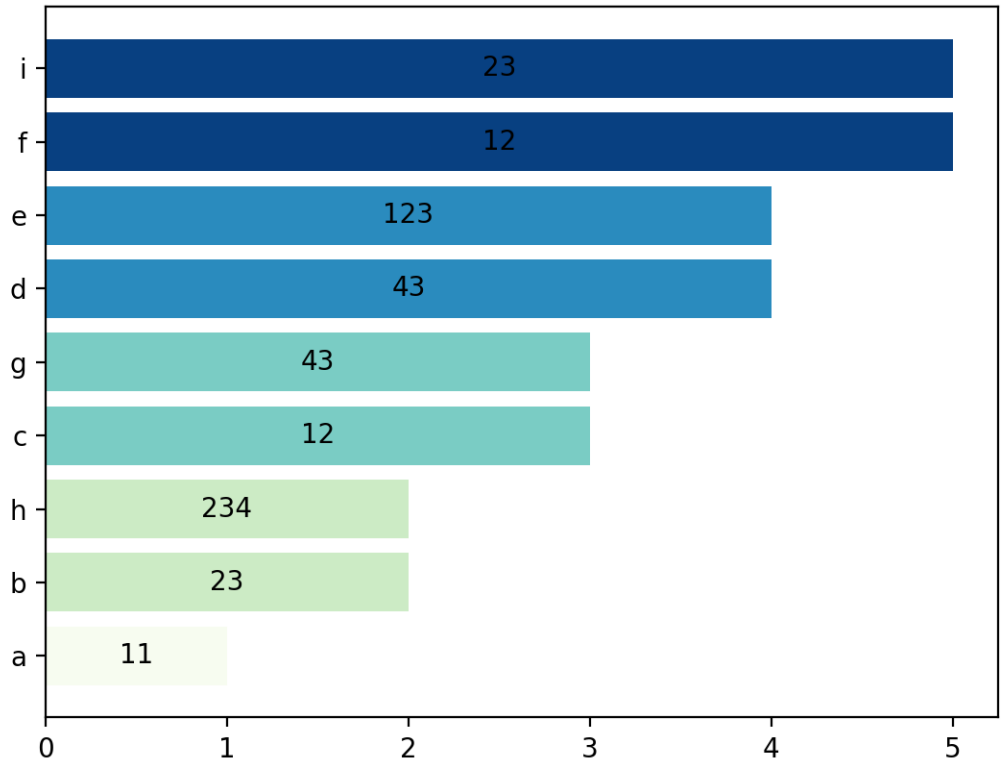
The default color of bars are chosen randomly. We can specify the color in many ways, e.g. a single color for all bars

```
_ = bar_chart([1,2,3,4,4,5,3,2,5], color="salmon")
```



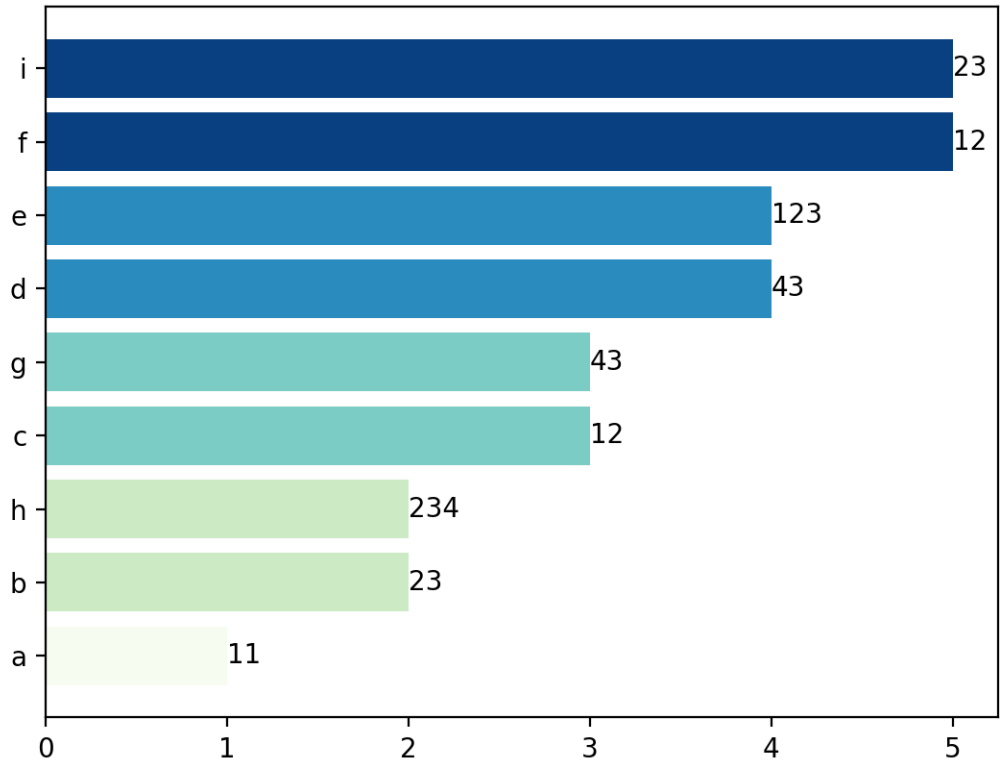
adding bar labels

```
_ = bar_chart(
    [1,2,3,4,4,5,3,2,5],
    ['a','b','c','d','e','f','g','h','i'],
    bar_labels=[11, 23, 12,43, 123, 12, 43, 234, 23],
    cmap="GnBu",
    sort=True)
```



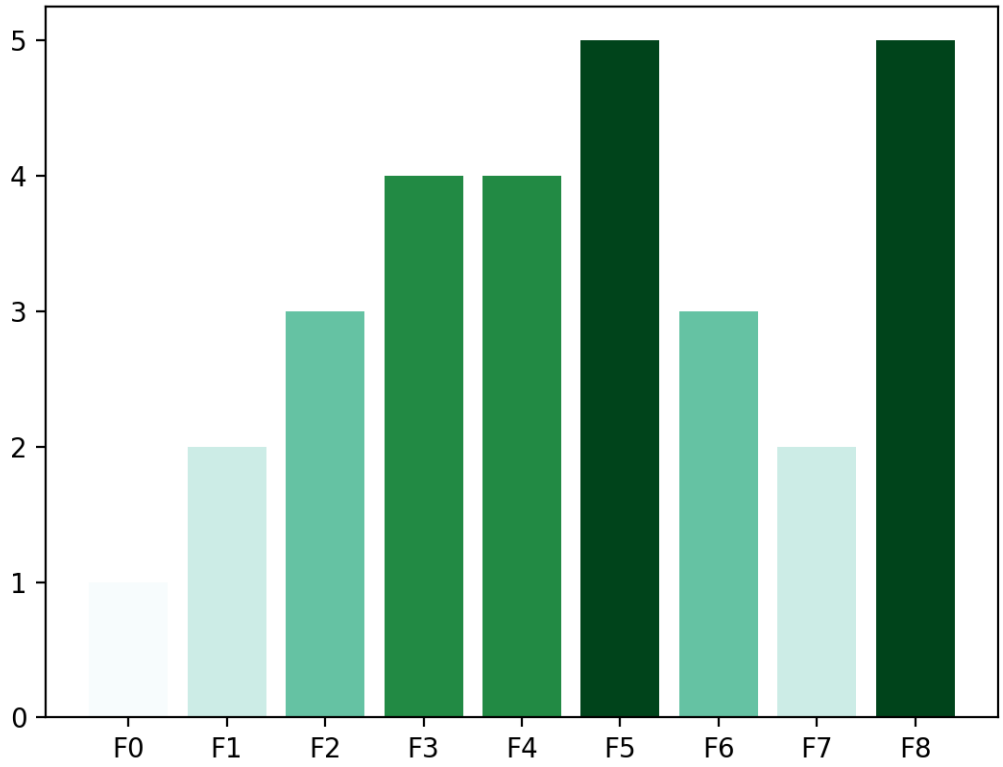
putting bar labels outside the bar

```
_ = bar_chart(
    [1,2,3,4,4,5,3,2,5],
    ['a','b','c','d','e','f','g','h','i'],
    bar_labels=[11, 23, 12,43, 123, 12, 43, 234, 23],
    bar_label_kws={'label_type':'edge'},
    cmap="GnBu",
    sort=True)
```



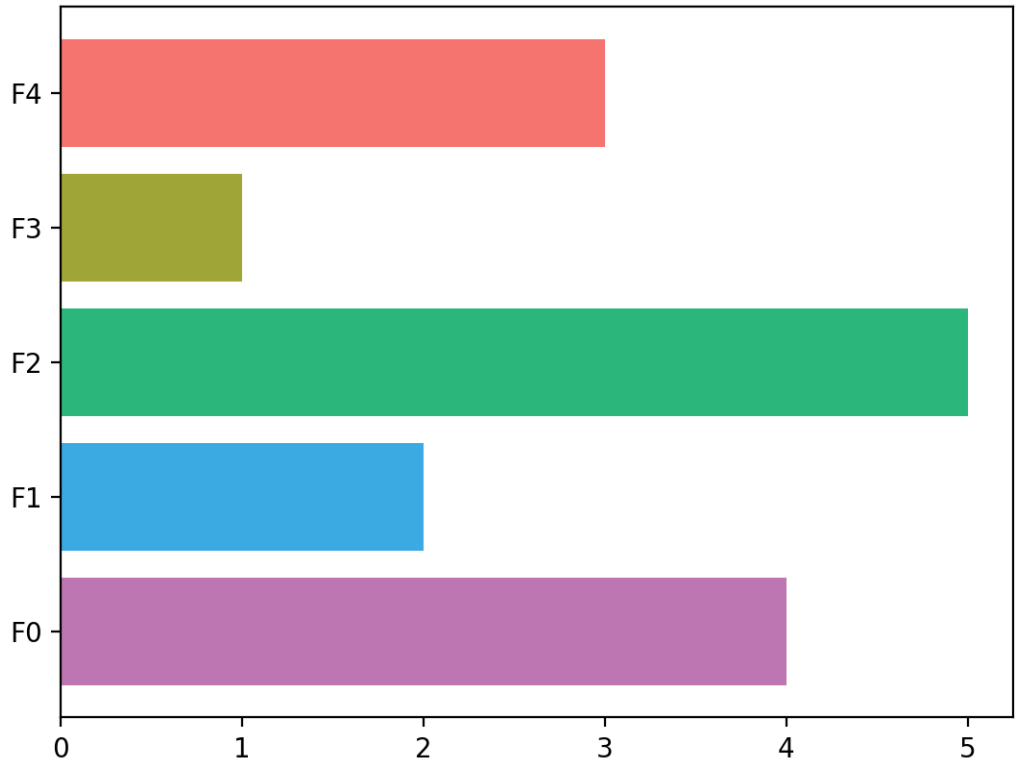
vertical orientation

```
_ = bar_chart([1,2,3,4,4,5,3,2,5], orient='v')
```



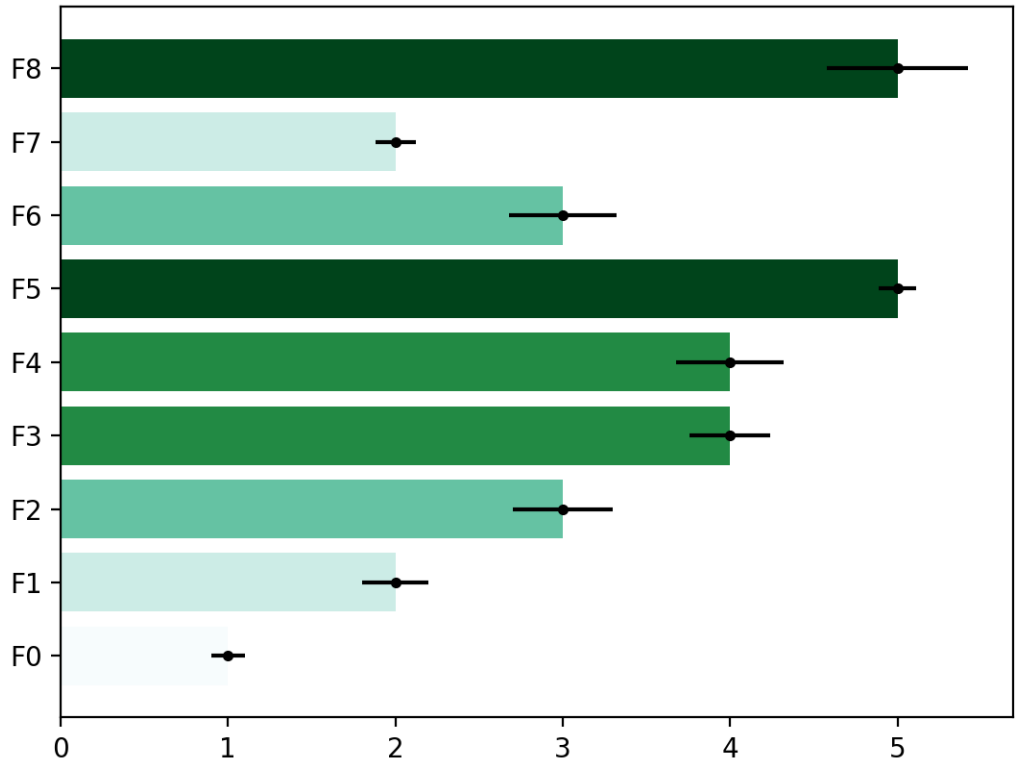
define color for each bar individually

```
_ = bar_chart([4,2,5,1,3], color=['#BD76B2', '#3BAAE2', '#2BB67B', '#9FA537', '#F5746F'])
```



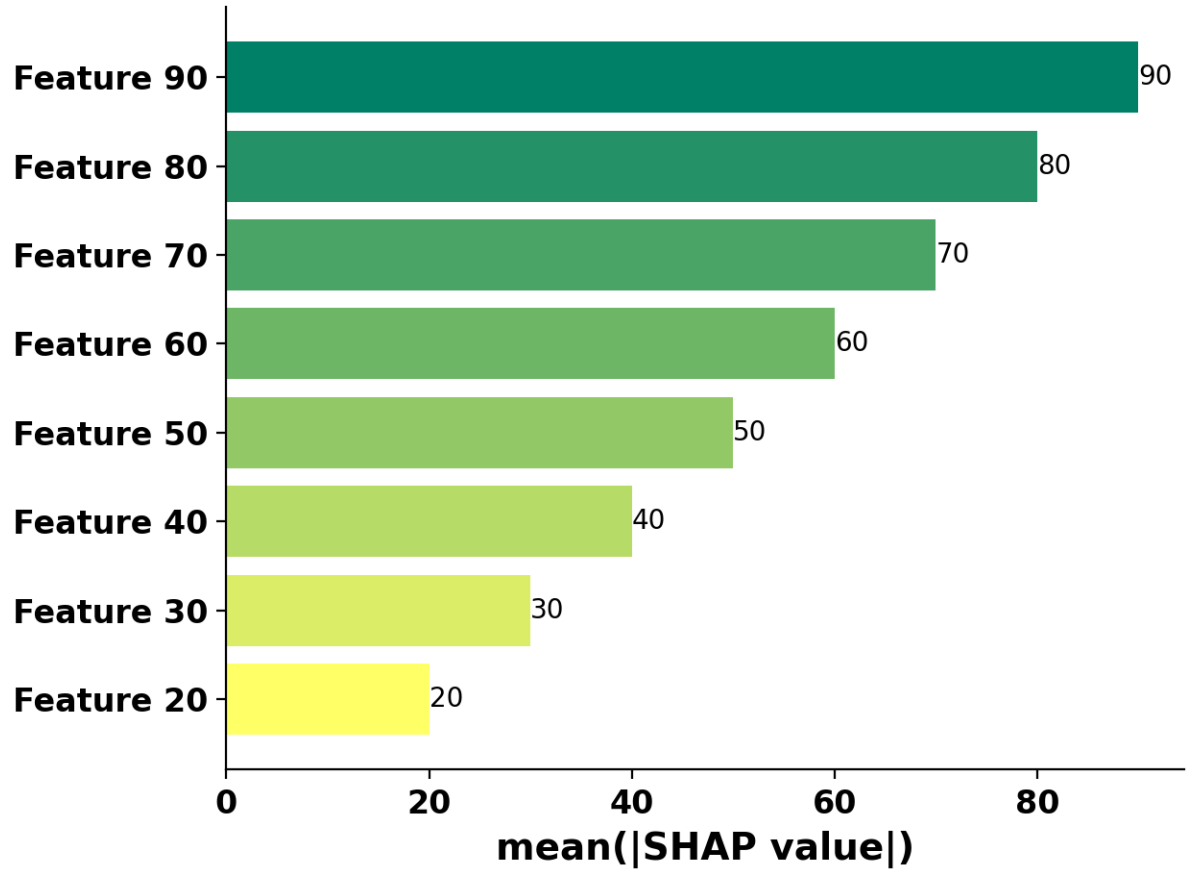
error bars

```
errors = [0.1, 0.2, 0.3, 0.24, 0.32, 0.11, 0.32, 0.12, 0.42]
_ = bar_chart([1,2,3,4,4,5,3,2,5], errors=errors)
```



the function bar_chart returns matplotlib axes which can be used for further processing

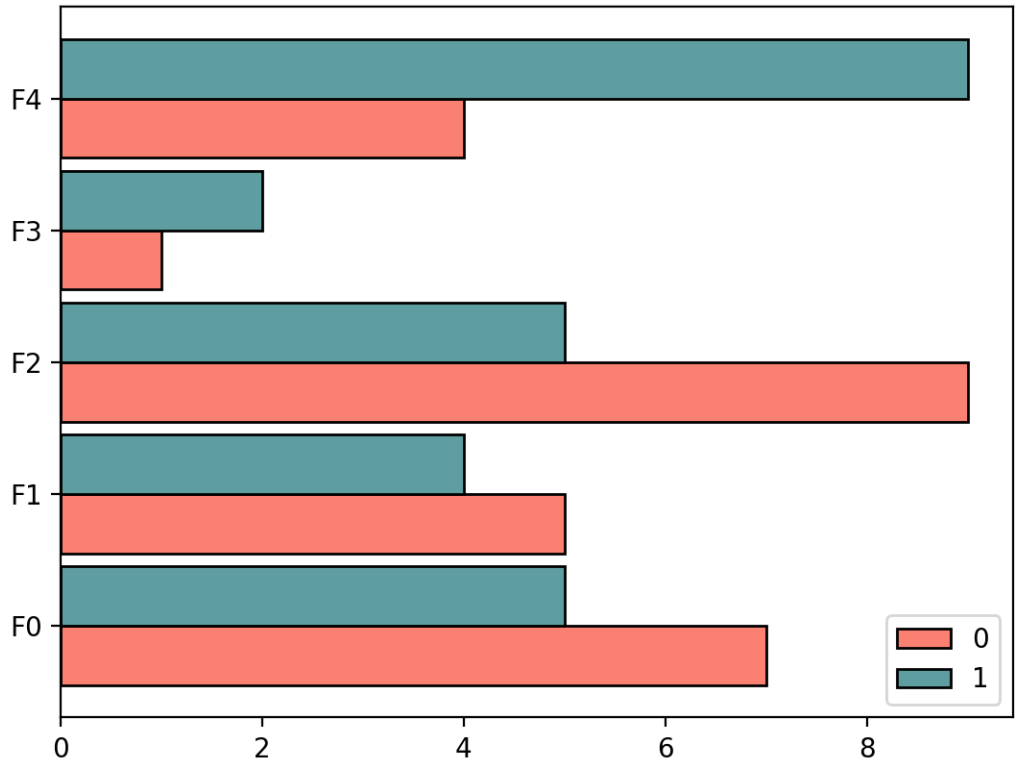
```
sv_bar = np.arange(20, 100, 10)
names = [f"Feature {n}" for n in sv_bar]

ax = bar_chart(sv_bar, names,
          bar_labels=sv_bar, bar_label_kws={'label_type':'edge'},
          show=False, sort=True, cmap='summer_r')

print(f"Type of ax is {type(ax)}")

despine_axes(ax, keep=['bottom', 'left'])
ax.set_xlabel(xlabel='mean(|SHAP value|)', fontsize=14, weight='bold')
ax.set_xticklabels(ax.get_xticks().astype(int), size=12, weight='bold')
ax.set_yticklabels(ax.get_yticklabels(), size=12, weight='bold')
plt.tight_layout()
plt.show()
```




```
Type of ax is <class 'matplotlib.axes._axes.Axes'>
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_bar.py:91: UserWarning: set_ticklabels() should only be used with a fixed␣
↪number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticks().astype(int), size=12, weight='bold')
```

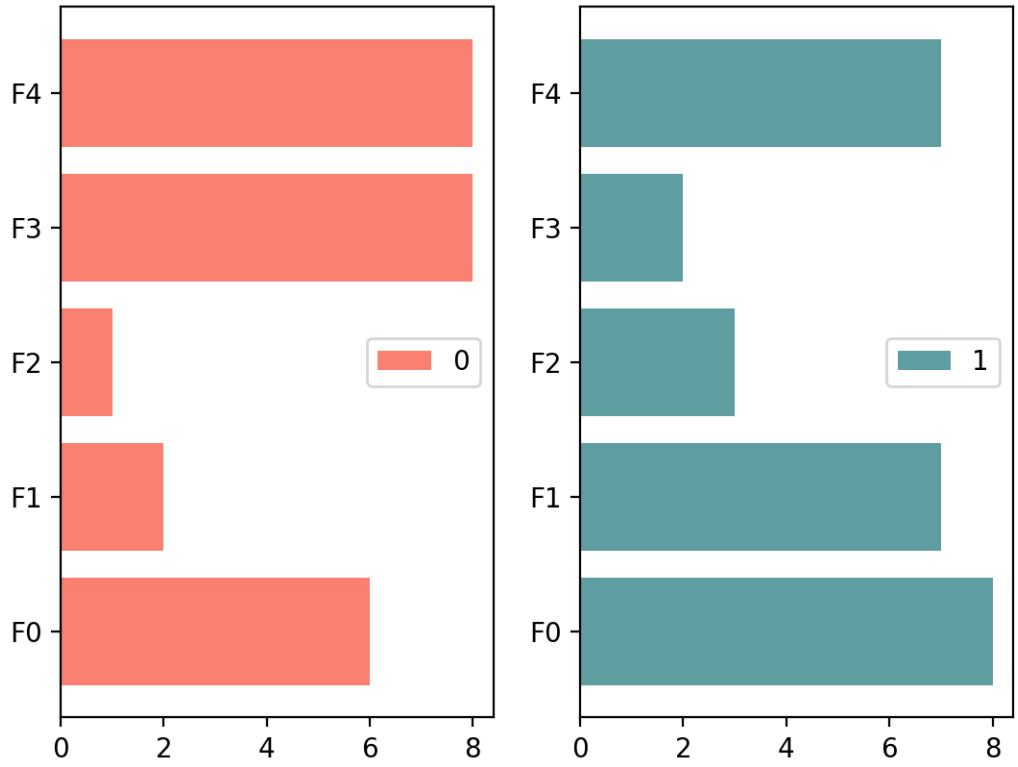
multipler bar charts

```
data = np.random.randint(1, 10, (5, 2))
_ = bar_chart(data, color=['salmon', 'cadetblue'])
```



multipler bar charts on separate axes

```
data = np.random.randint(0, 10, (5, 2))
_ = bar_chart(data, color=['salmon', 'cadetblue'], share_axes=False)
```



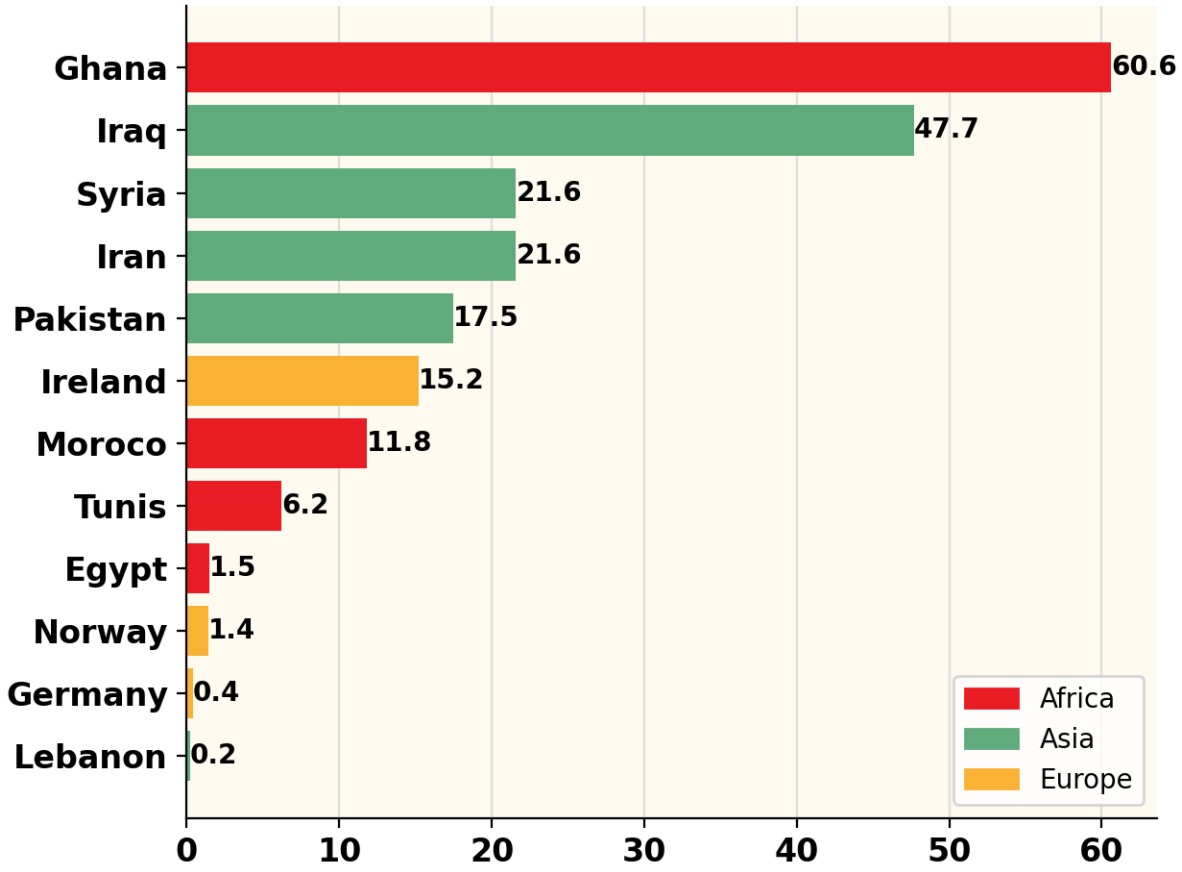
specifying colors for group of bars

```
colors = {'Asia': '#60AB7B',
          'Europe': '#F9B234',
          'Africa': '#E91B23'}

continents = {'Pakistan': 'Asia', 'Iran': 'Asia', 'Syria': 'Asia',
         'Iraq': 'Asia', 'Lebanon': 'Asia', 'Ireland': 'Europe',
         'Germany': 'Europe', 'Norway': 'Europe', 'Ghana': 'Africa',
         'Egypt': 'Africa', 'Moroco': 'Africa', 'Tunis': 'Africa'}

data = [ 17.5,  21.6,  21.6,  47.7,
         0.2,  15.2,   0.4 ,   1.4,
         60.6,   1.5,  11.8,   6.2]

ax = bar_chart(data, list(continents.keys()),
          bar_labels=data, bar_label_kws={'label_type':'edge',
                                          'fontsize': 10,
                                          'weight': 'bold'},
          show=False, sort=True, color=[colors[val] for val in continents.values()],
          ax_kws=dict(top_spine=False, right_spine=False))

ax.set_xticklabels(ax.get_xticks().astype(int), size=12, weight='bold')
ax.set_yticklabels(ax.get_yticklabels(), size=12, weight='bold')
```

(continues on next page)

(continued from previous page)

```
labels = np.unique(list(continents.values()))
handles = [plt.Rectangle((0,0),1,1, color=colors[l]) for l in labels]
ax.xaxis.grid(True, linestyle='-', which='major', color='lightgrey',
              alpha=0.7)
ax.set(axisbelow=True)  # Hide the grid behind plot objects
ax.set_facecolor('floralwhite')
plt.legend(handles, labels, loc='lower right')
plt.tight_layout()
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_bar.py:129: UserWarning: set_ticklabels() should only be used with a fixed␣
↪number of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels(ax.get_xticks().astype(int), size=12, weight='bold')
```

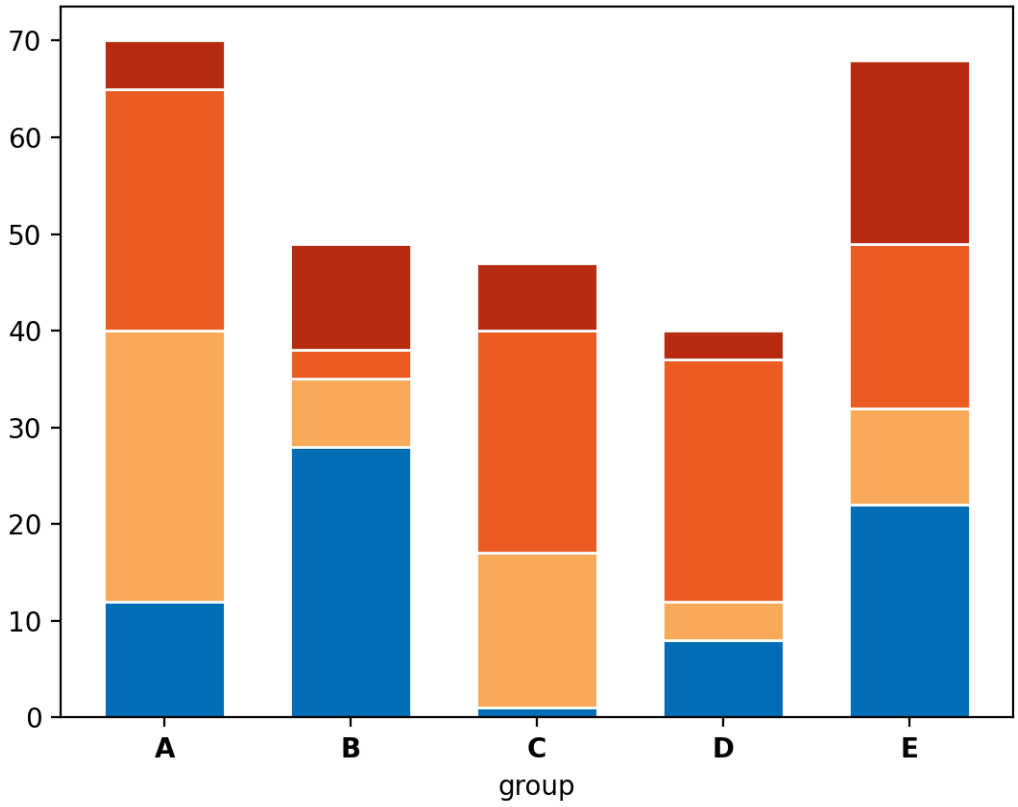
Stacked bar chart

```
# Values of each group
bars1 = [12, 28, 1, 8, 22]
bars2 = [28, 7, 16, 4, 10]
bars3 = [25, 3, 23, 25, 17]
bars4 = [5, 11, 7, 3, 19]
```

(continues on next page)

(continued from previous page)

```
# Heights of bars1 + bars2
bars = np.add(bars1, np.add(bars2, bars3).tolist()).tolist()

# Names of group and bar width
names = ['A', 'B', 'C', 'D', 'E']
barWidth = 0.65

bar_chart(bars1, color='#006db6', edgecolor='white',
          width=barWidth, orient='v', show=False)

bar_chart(bars2, bottom=bars1, color='#f8aa59',
          edgecolor='white', width=barWidth, orient='v',
          show=False)

bar_chart(bars3, bottom=np.add(bars1, bars2).tolist(), color='#eb5c23',
          edgecolor='white', width=barWidth, orient='v',
          show=False)

bar_chart(bars4, bottom=bars, color='#b72c10',
          edgecolor='white', width=barWidth, orient='v',
          show=False)

# Custom X axis
plt.xticks([0, 1, 2, 3, 4], names, fontweight='bold')
plt.xlabel("group")
plt.show()
```



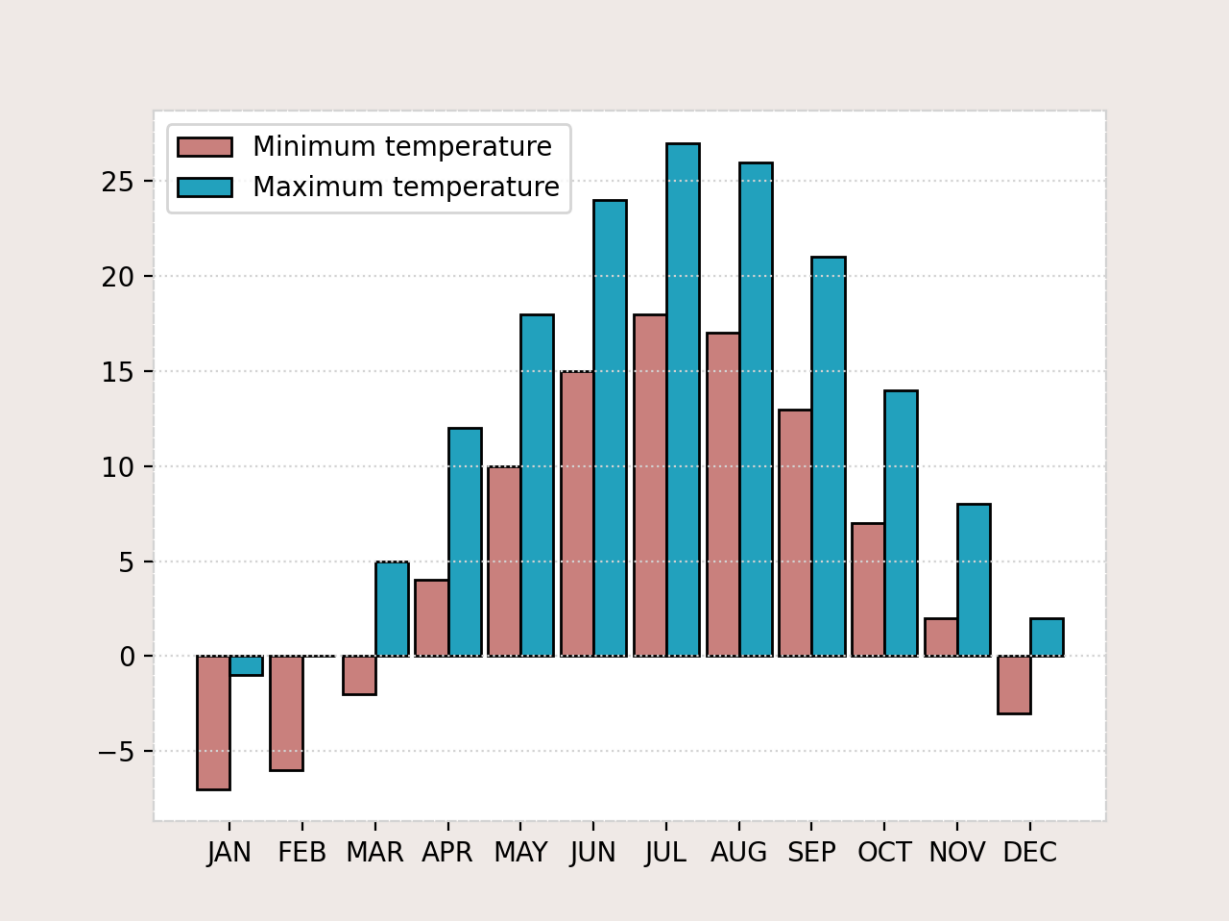
negative values in the data

```
names = ['JAN','FEB','MAR','APR','MAY','JUN',
         'JUL','AUG','SEP','OCT','NOV','DEC']

temp_max = [-1,0,5,12,18,24,27,26,21,14,8,2]
temp_min = [-7,-6,-2,4,10,15,18,17,13,7,2,-3]


def listOfTuples(l1, l2):
    return list(map(lambda x, y:(x,y), l1, l2))

temp = listOfTuples(temp_min, temp_max)

f, ax = plt.subplots(facecolor = "#EFE9E6")

bar_chart(temp, color=['#c9807d', '#22a1bd'],
              orient='v', labels=names, ax=ax,
              show=False)

ax.grid(axis='y', ls='dotted', color='lightgrey')

for spine in ax.spines.values():
    spine.set_edgecolor('lightgrey')
```

(continues on next page)

(continued from previous page)

```
    spine.set_linestyle('dashed')

plt.legend(['Minimum temperature', 'Maximum temperature'])
plt.show()
```



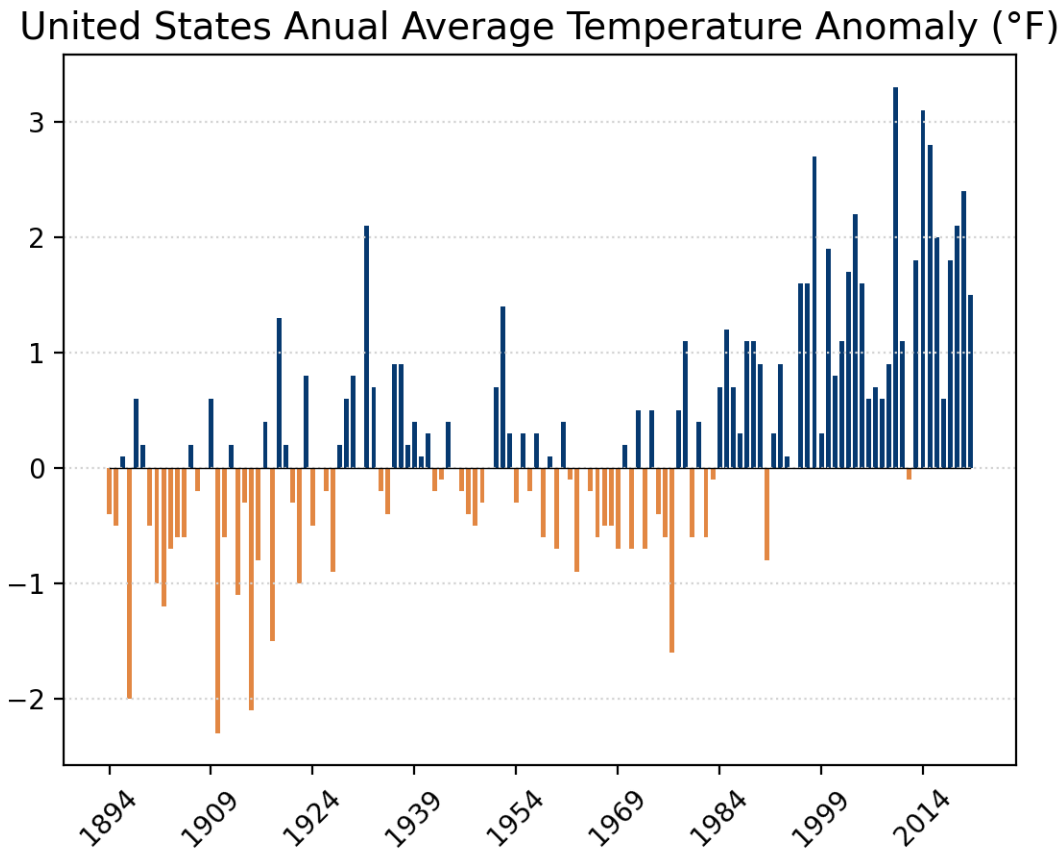
Displaying nagative values with a specific color

```
data = [-0.4, -0.5, 0.1, -2, 0.6, 0.2, -0.5, -1, -1.2,
-0.7, -0.6, -0.6, 0.2, -0.2, 0, 0.6, -2.3, -0.6, 0.2, -1.1, -0.3, -2.1, -0.8, 0.4,
-1.5, 1.3, 0.2, -0.3, -1, 0.8, -0.5, 0, -0.2, -0.9, 0.2, 0.6, 0.8, 0, 2.1, 0.7, -0.2,
-0.4, 0.9, 0.9, 0.2, 0.4, 0.1, 0.3, -0.2, -0.1, 0.4, 0, -0.2, -0.4, -0.5, -0.3, 0, 0.7,
1.4, 0.3, -0.3, 0.3, -0.2, 0.3, -0.6, 0.1, -0.7, 0.4, -0.1, -0.9, 0, -0.2, -0.6, -0.5,
-0.5, -0.7, 0.2, -0.7, 0.5, -0.7, 0.5, -0.4, -0.6, -1.6, 0.5, 1.1, -0.6, 0.4, -0.6, -0.1,
0.7, 1.2, 0.7, 0.3, 1.1, 1.1, 0.9, -0.8, 0.3, 0.9, 0.1, 0, 1.6, 1.6, 2.7, 0.3, 1.9, 0.8,
1.1, 1.7, 2.2, 1.6, 0.6, 0.7, 0.6, 0.9,3.3, 1.1, -0.1, 1.8, 3.1, 2.8, 2, 0.6, 1.8,
2.1, 2.4, 1.5]

colors = ['#063970' if e >= 0 else '#e28743' for e in data]

ax = bar_chart(data, orient='v', color=colors,
          width=0.7, rotation=45,
          labels='', show=False)
plot(np.zeros(128), ax=ax, show=False, color='black', lw=0.5)
```

(continues on next page)

(continued from previous page)

```
ax.grid(axis='y', ls='dotted', color='lightgrey')

times = np.arange(np.datetime64('1894'),
                  np.datetime64('2022'), np.timedelta64(1, 'Y'))

ax.set_xticklabels(times)
ax.xaxis.set_major_locator(plt.MaxNLocator(10))
ax.set_title('United States Anual Average Temperature Anomaly (°F)',
             fontdict={'fontsize': 14})
plt.show()
```



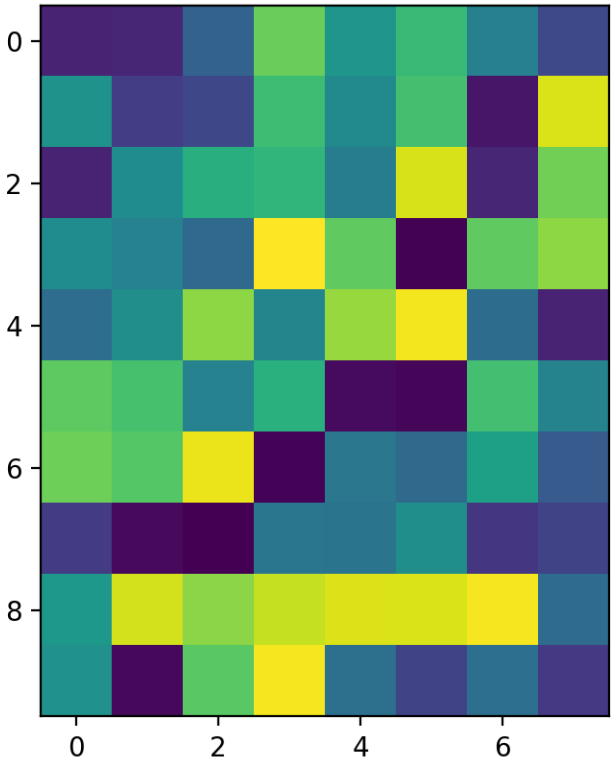
Total running time of the script: (0 minutes 4.675 seconds)

6.4 heatmap/imshow

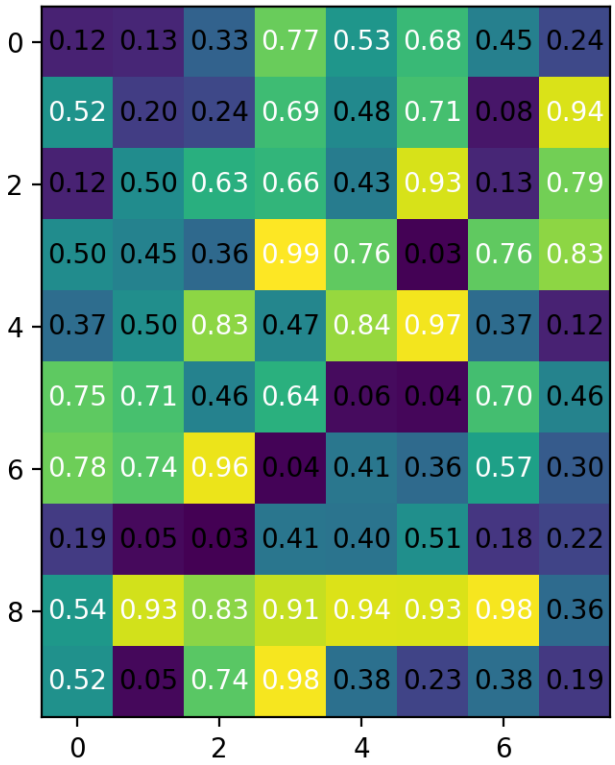
```
# sphinx_gallery_thumbnail_number = 11

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from easy_mpl import imshow
from easy_mpl.utils import version_info, despine_axes

version_info()  # print version information of all the packages being used
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
x = np.random.random((10, 8))

_ = imshow(x)
```

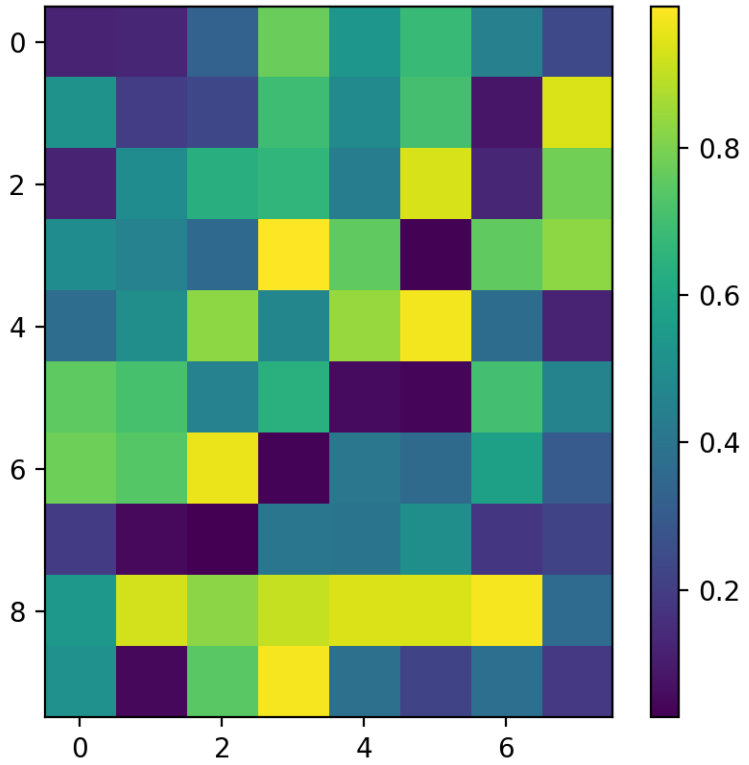



```
_ = imshow(x, annotate=True)
```



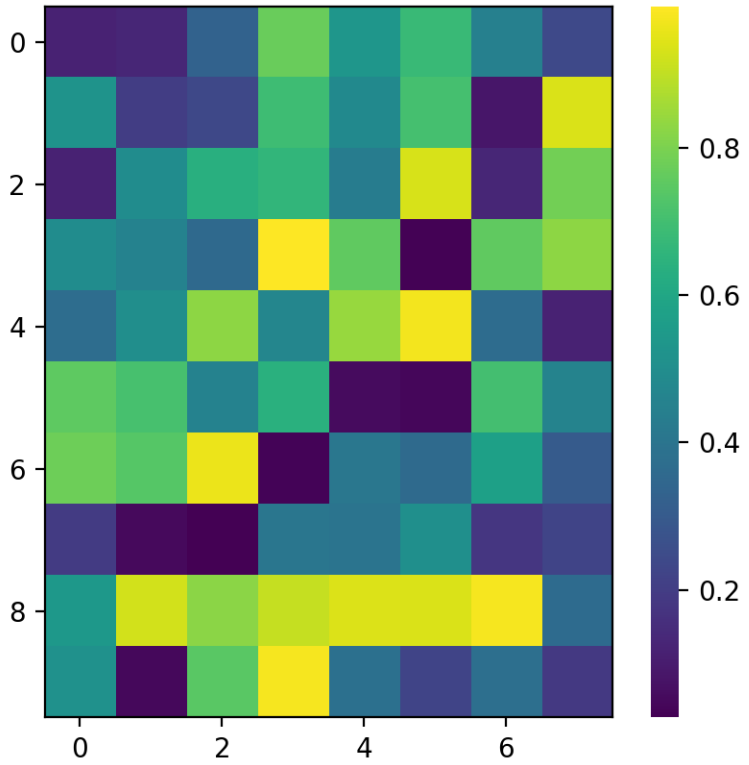
show colorbar

```
_ = imshow(x, colorbar=True)
```



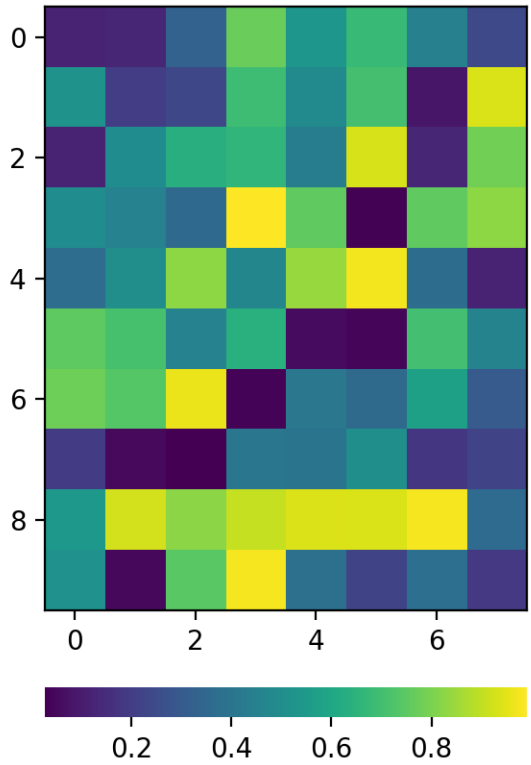
do not show border around colorbar

```
_ = imshow(x, colorbar=True, cbar_params={"border": False})
```



Move the colorbar below the heatmap

```
_ = imshow(x, colorbar=True, cbar_params={"border": False, 'pad': 0.4,
                                          "orientation":"horizontal"})
```

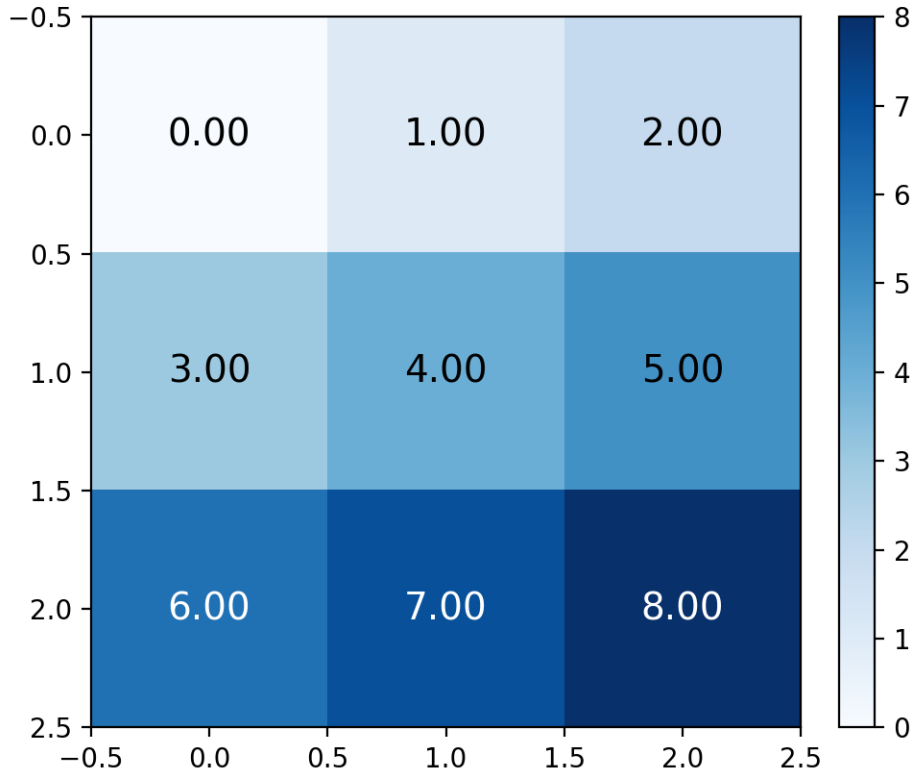


show white grid line

```
data = np.random.random((4, 10))

_ = imshow(data, cmap="YlGn",
       xticklabels=[f"Feature {i}" for i in range(data.shape[1])],
       grid_params={'border': True, 'color': 'w', 'linewidth': 2},
           annotate=True,
       colorbar=True)
```

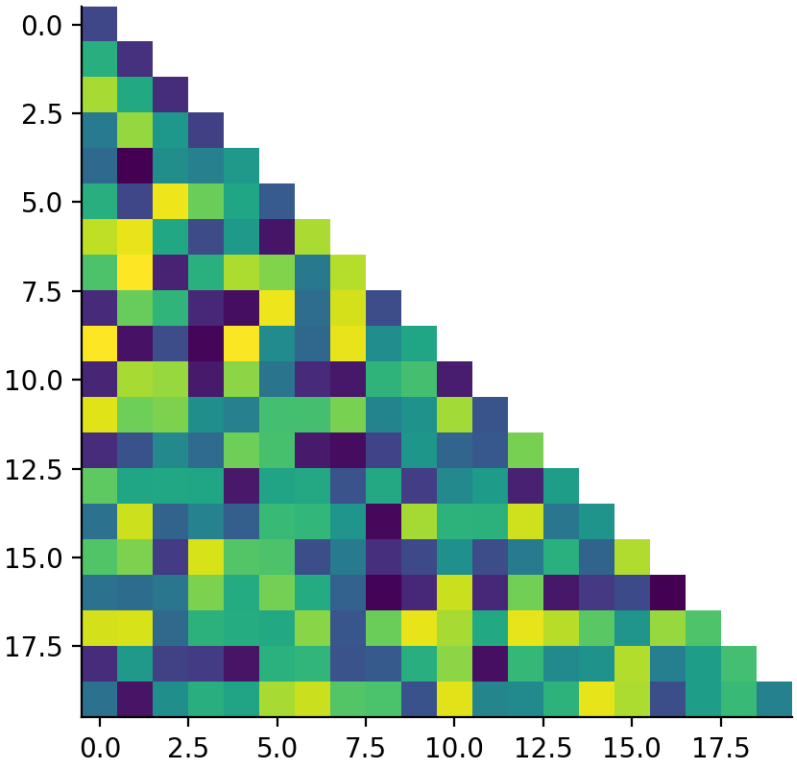
we can specify color of text in each box of imshow for annotation For this, `textcolors` must a numpy array of shape same as that of data. Each value in this numpy array will define color for corresponding box annotation.

```
data = np.arange(9).reshape((3,3))

_ = imshow(data, cmap="Blues",
       annotate=True,
       annotate_kws={
              "textcolors": np.array([['black', 'black', 'black'],
                                      ['black', 'black', 'black'],
                                     ['white', 'white', 'white']]),
              'fontsize': 14
       },
       colorbar=True)
```

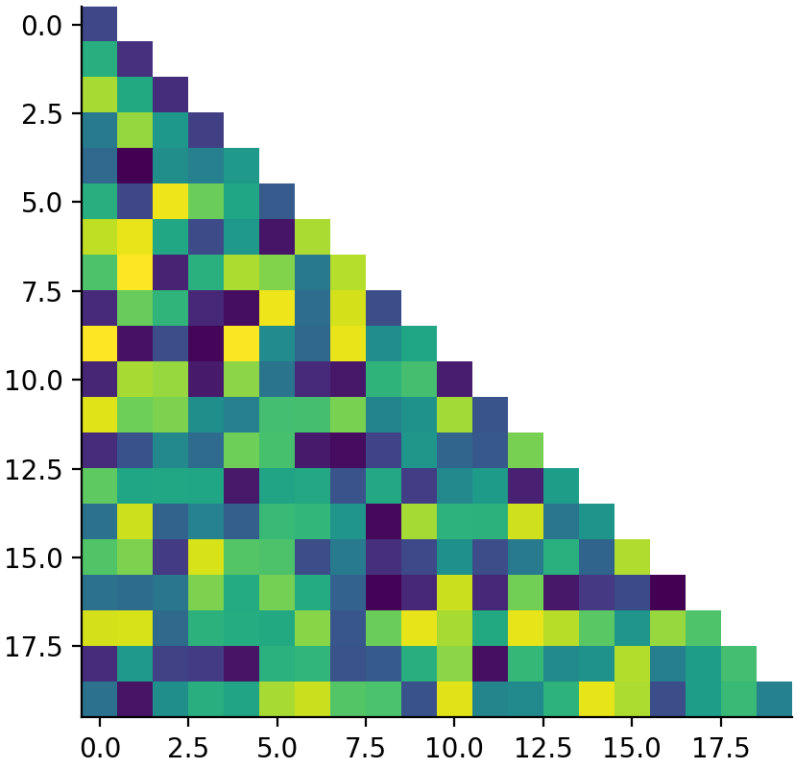


We can decide which portion of heatmap to show using `mask` argument

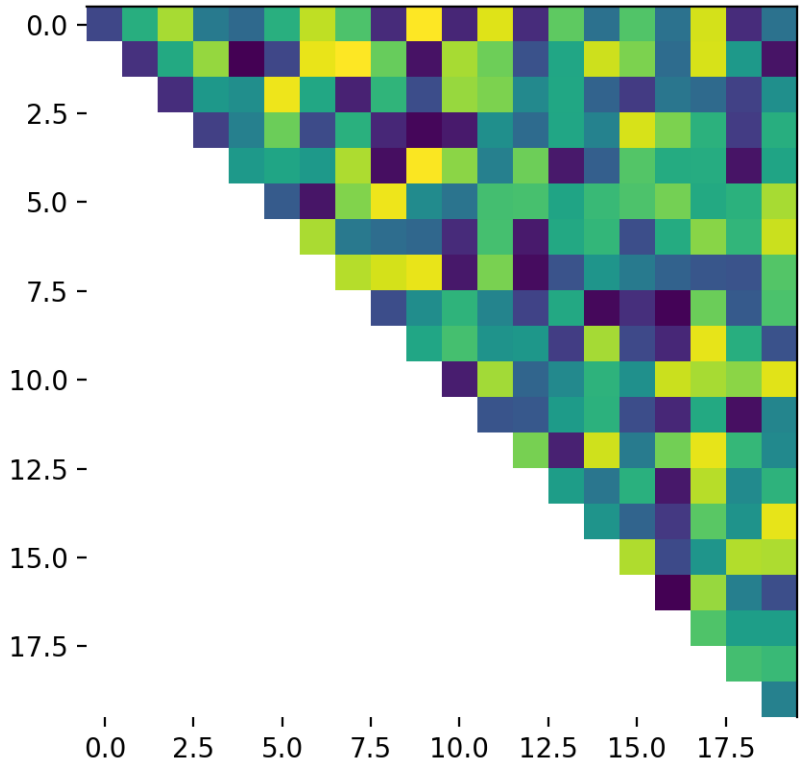
```
x = np.random.random((20, 20))
_ = imshow(x, mask=True)
```




```
_ = imshow(x, mask="upper")
```




```
_ = imshow(x, mask="lower")
```



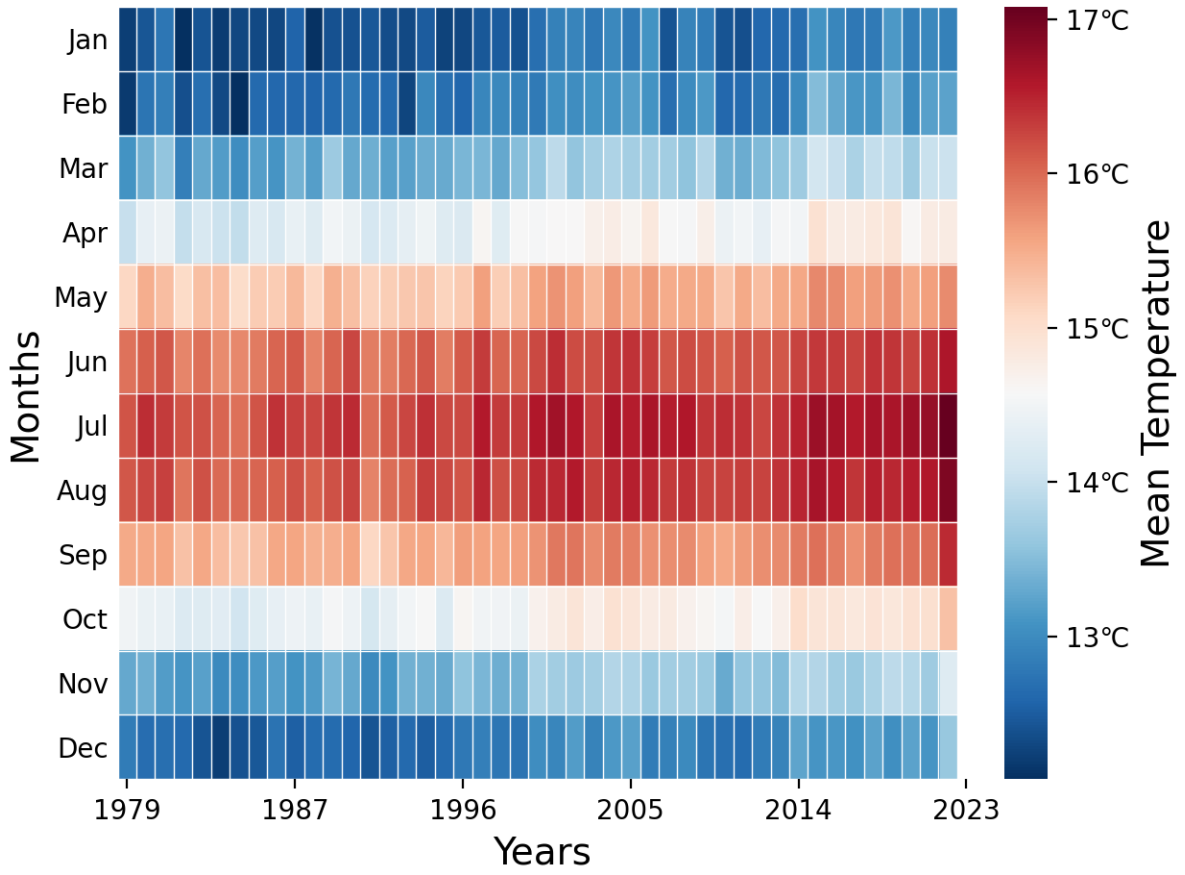
The imshow function returns Axesimage object of matplotlib which can be used for further processing. The Axesimage is not axes, but we can get the axes from Axesimage using Axesimage.axes the process it as shown below.

```
data = pd.read_json('https://climatereanalyzer.org/clim/t2_daily/json_cfsr/cfsr_world_t2_
↪day.json')
index = data.pop('name')
nyrs = 45
data = pd.DataFrame(
    np.array([np.array(data.iloc[row, :].values[0]) for row in range(nyrs)]),
    index=pd.to_datetime(index[0:nyrs])
)
data = data.astype(float)
data1 = pd.concat([data.iloc[i, :] for i in range(data.shape[0])]).dropna()
data1.index = pd.date_range(data.index[0], periods=len(data1), freq="D")
mon_data = data1.resample('M').mean()

data_np = np.full(shape=(12, nyrs), fill_value=np.nan)
for ii, i in enumerate(range(0, len(mon_data), 12)):
    data_np[:, ii] = mon_data.iloc[i:i + 12].values

print(data_np.shape)

im = imshow(
    data_np,
```

(continues on next page)

(continued from previous page)

```
    cmap="RdBu_r",
    aspect="auto",
    colorbar=True,
    cbar_params=dict(border=False, title="Mean Temperature",
                     title_kws=dict(fontsize=14)),
    show=False,
    ax_kws=dict(xlabel="Years", ylabel="Months",
                xlabel_kws=dict(fontsize=14), ylabel_kws=dict(fontsize=14)),
    grid_params={'border': True, 'color': 'w', 'linewidth': 0.5},
)
im.axes.set_yticks(range(12))
im.axes.set_yticklabels(
    ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'])
im.axes.set_xticks(np.linspace(0, data_np.shape[-1], 6))
im.axes.set_xticklabels(np.linspace(data.index.year.min(), data.index.year.max(), 6,␣
↪dtype=int))
despine_axes(im.axes)
im.axes.tick_params(axis=u'y', which=u'both',length=0)
ticklabels = []
for ticklabel in im.colorbar.ax.get_yticklabels():
    ticklabel.set_text(f"{ticklabel.get_text()}℃")
    ticklabels.append(ticklabel)
im.colorbar.set_ticklabels(ticklabels)
plt.tight_layout()
plt.show()
```




```
(12, 45)
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_imshow.py:131: UserWarning: set_ticklabels() should only be used with a␣
↪fixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
  im.colorbar.set_ticklabels(ticklabels)
```

We can pass any valid matplotlib cmap. For example, we can use cmaps from seaborn library.

```
import seaborn as sns

cm = sns.color_palette("rocket_r", as_cmap=True)

print(type(cm))
```

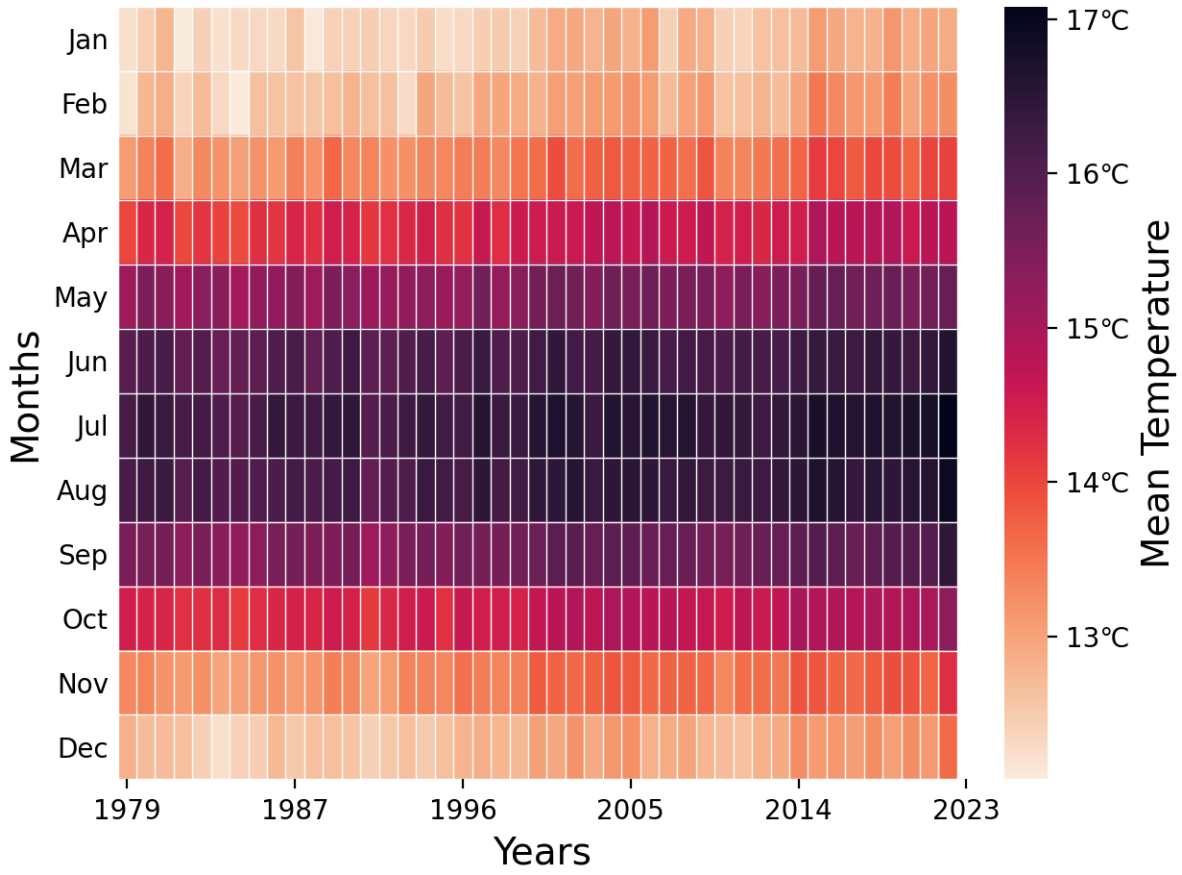
```
<class 'matplotlib.colors.ListedColormap'>
```

```
im = imshow(
    data_np,
    cmap=cm,
    aspect="auto",
    colorbar=True,
    cbar_params=dict(border=False, title="Mean Temperature",
                     title_kws=dict(fontsize=14)),
```

(continues on next page)

(continued from previous page)

```
    show=False,
    ax_kws=dict(xlabel="Years", ylabel="Months",
                xlabel_kws=dict(fontsize=14), ylabel_kws=dict(fontsize=14)),
    grid_params={'border': True, 'color': 'w', 'linewidth': 0.5},
)
im.axes.set_yticks(range(12))
im.axes.set_yticklabels(
    ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'])
im.axes.set_xticks(np.linspace(0, data_np.shape[-1], 6))
im.axes.set_xticklabels(np.linspace(data.index.year.min(), data.index.year.max(), 6,␣
↪dtype=int))
despine_axes(im.axes)
im.axes.tick_params(axis=u'y', which=u'both',length=0)
ticklabels = []
for ticklabel in im.colorbar.ax.get_yticklabels():
    ticklabel.set_text(f"{ticklabel.get_text()}℃")
    ticklabels.append(ticklabel)
im.colorbar.set_ticklabels(ticklabels)
plt.tight_layout()
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_imshow.py:171: UserWarning: set_ticklabels() should only be used with a␣
↪fixed number of ticks, i.e. after set_ticks() or using a FixedLocator.
```

(continues on next page)

(continued from previous page)

```
  im.colorbar.set_ticklabels(ticklabels)
```

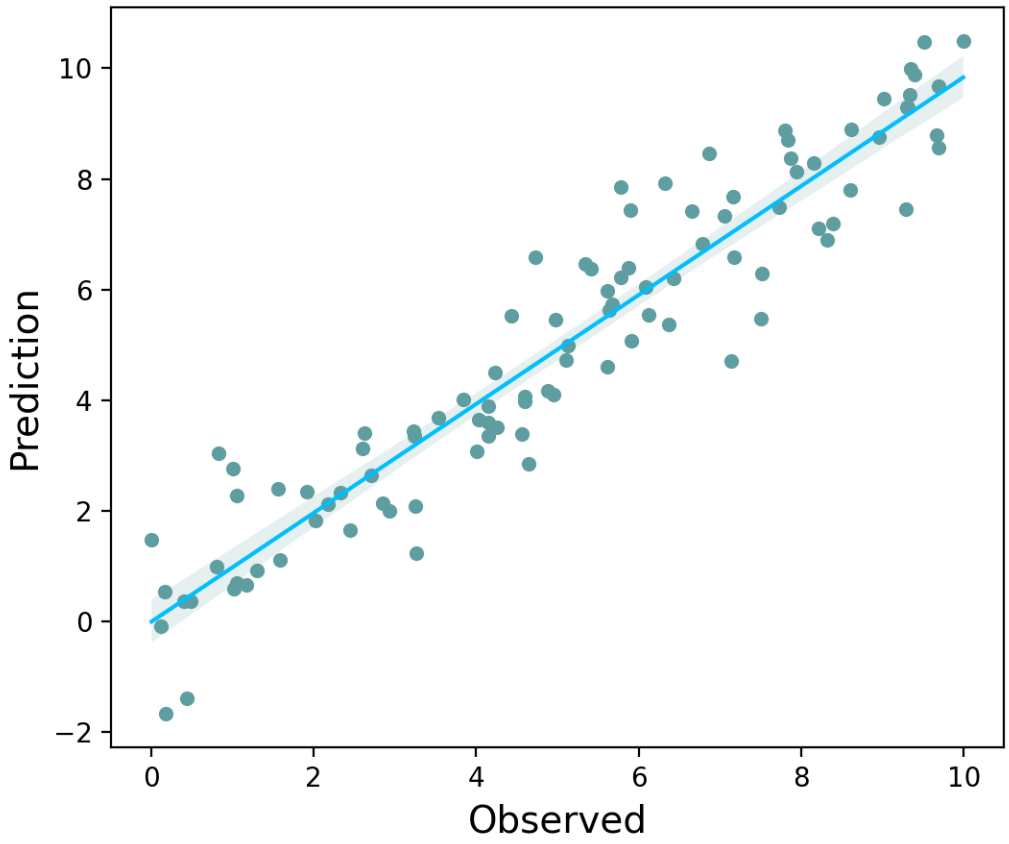
Total running time of the script: (0 minutes 5.027 seconds)

6.5 regression plot

```
# sphinx_gallery_thumbnail_number = -2

import numpy as np
from easy_mpl import regplot
import matplotlib.pyplot as plt
from easy_mpl.utils import version_info

version_info()
```

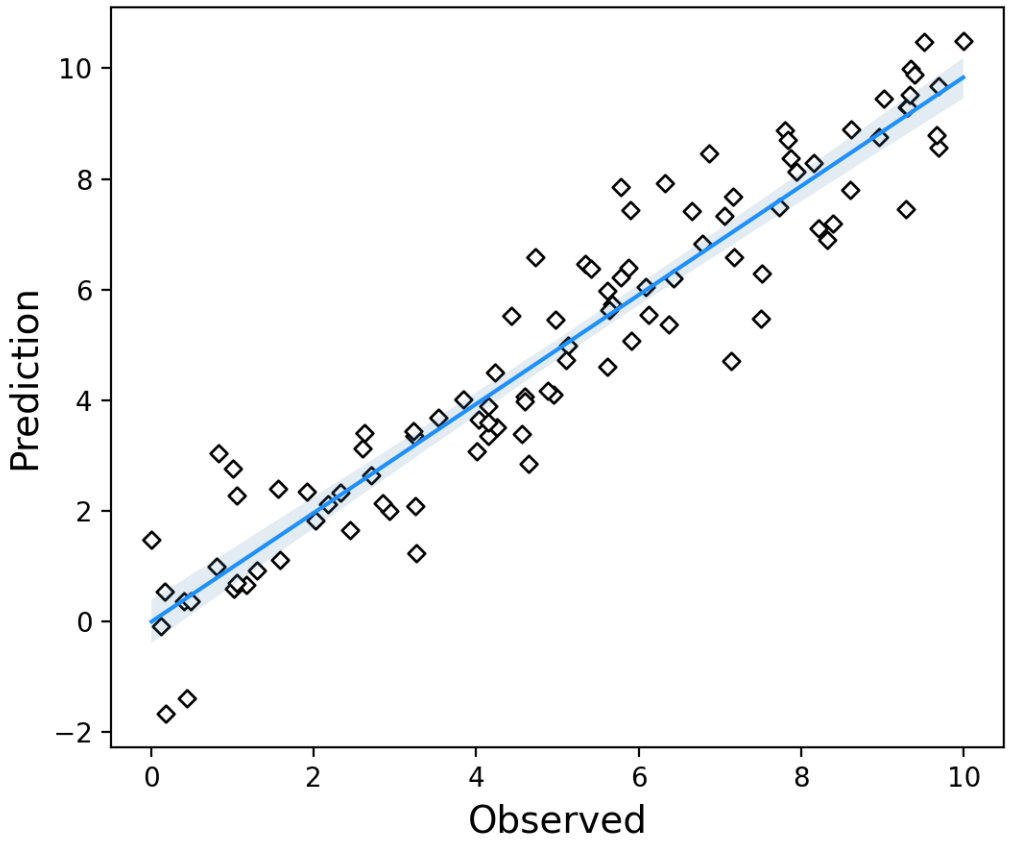
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
rng = np.random.default_rng(313)

x = rng.uniform(0, 10, size=100)
y = x + rng.normal(size=100)
_ = regplot(x, y)
```



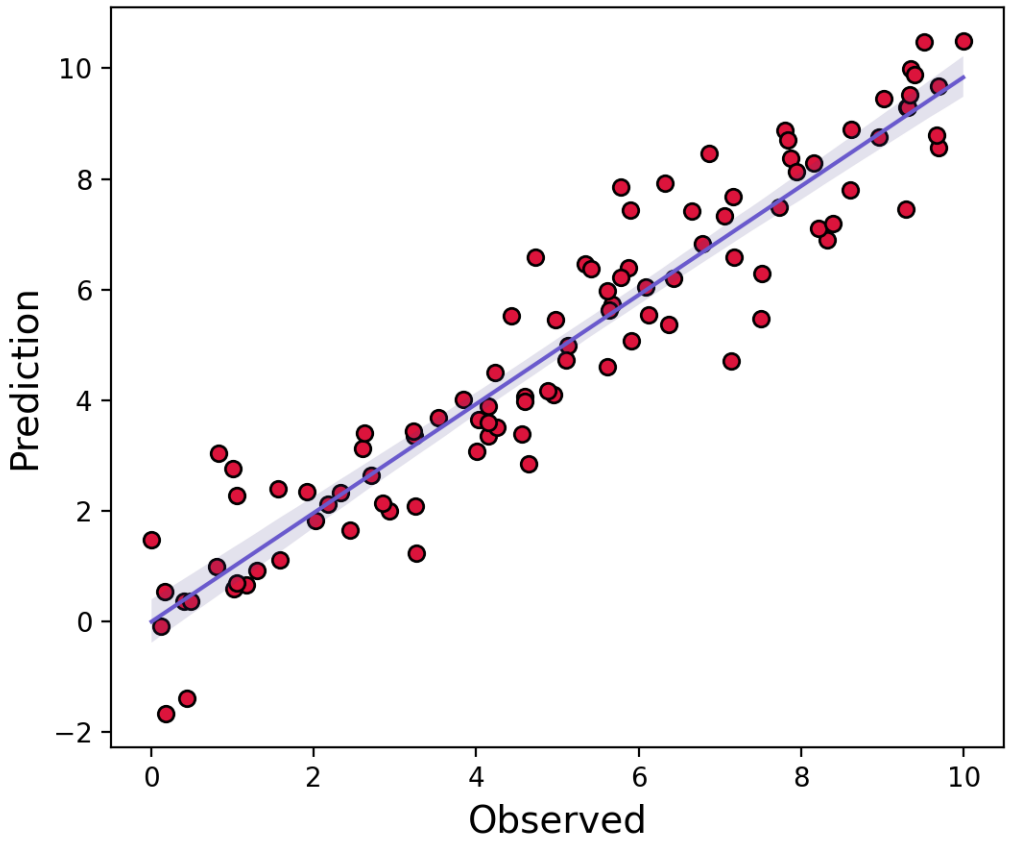
customizing marker style

```
_ = regplot(x, y, marker_color='white',
            scatter_kws={'marker':"D", 'edgecolors':'black'})
```



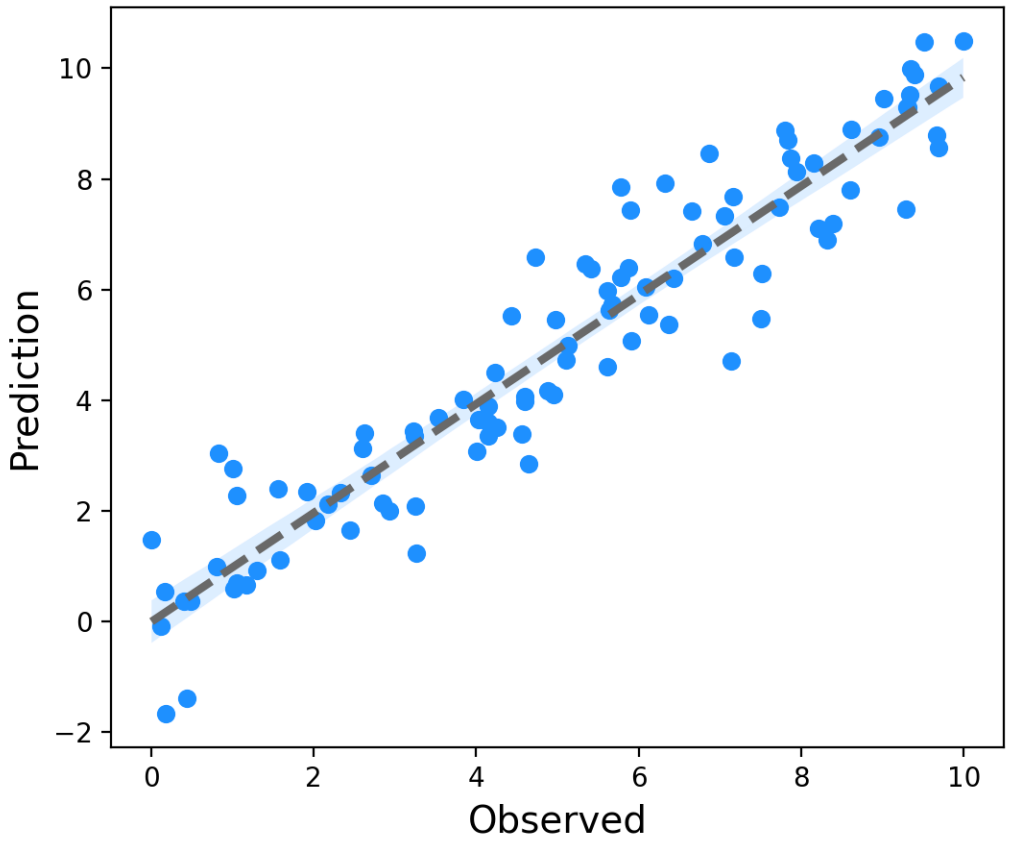
another example by increasing the *marker size*

```
_ = regplot(x, y, marker_color='crimson', marker_size=35,
            scatter_kws={'marker':"o", 'edgecolors':'black'})
```



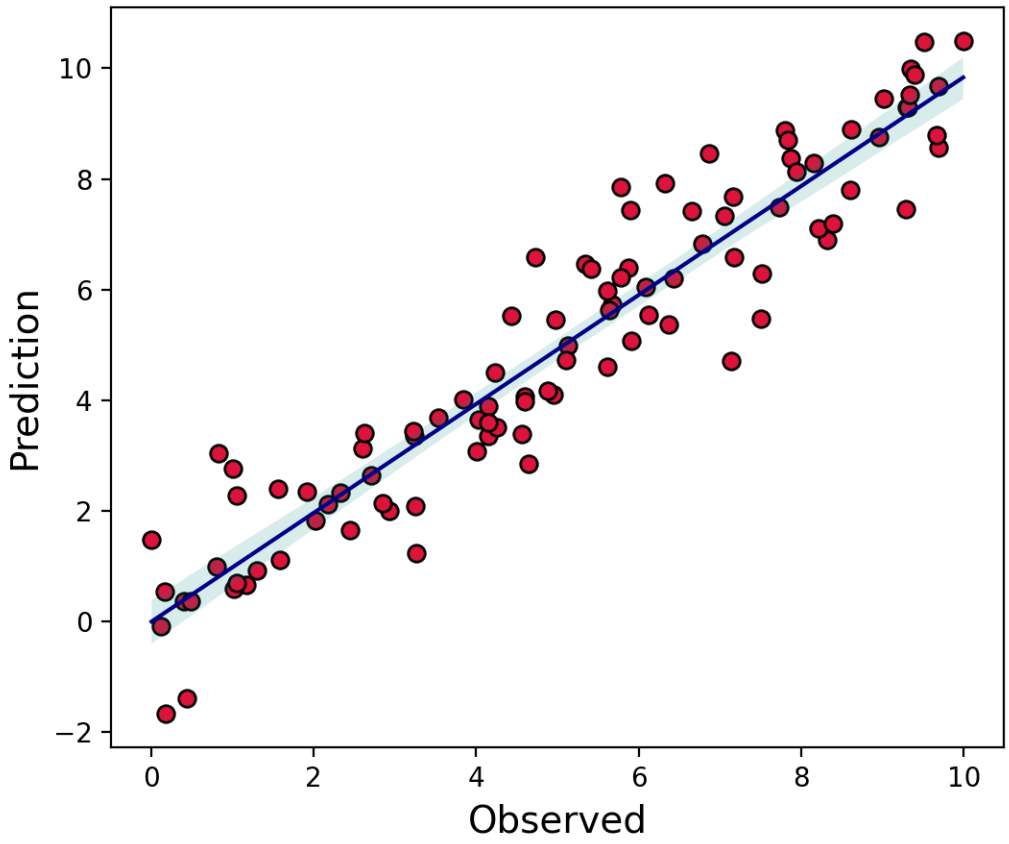
customizing line style

```
_ = regplot(x, y, marker_color='dodgerblue', marker_size=35,
            scatter_kws={'marker':"o"},
             line_color='dimgrey', line_style='--',
             line_kws={'linewidth':3})
```



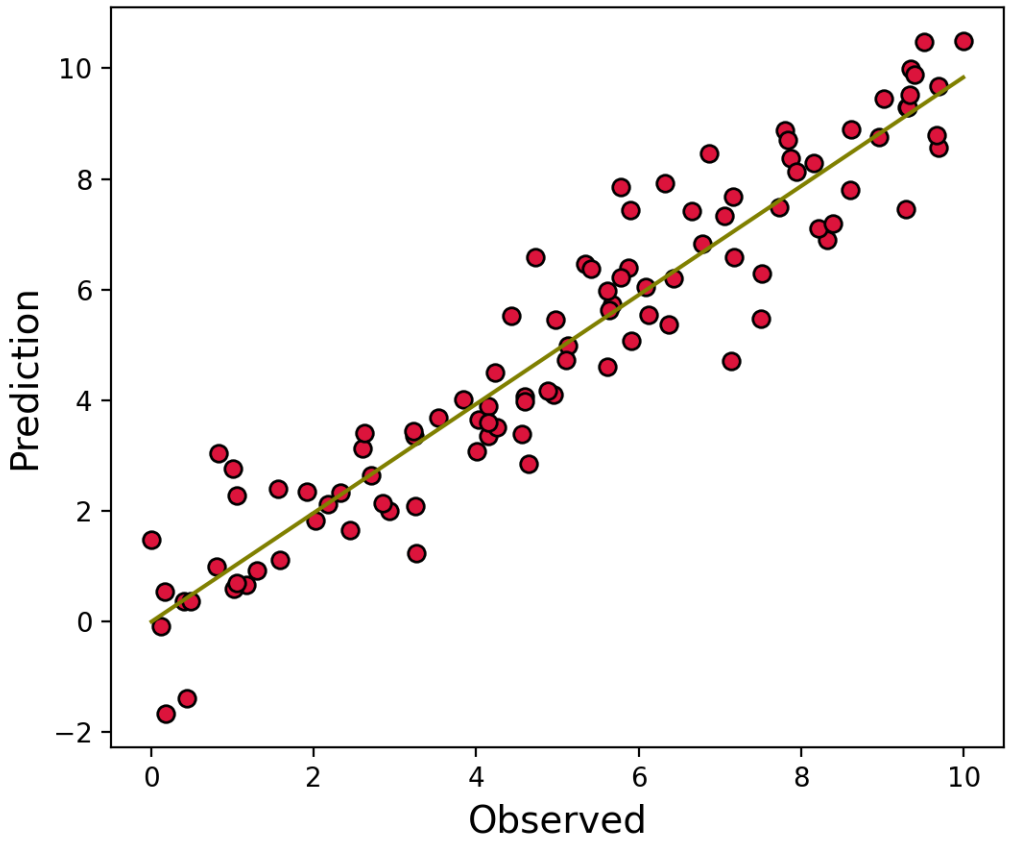
customizing fill color

```
_ = regplot(x, y, marker_color='crimson', marker_size=40,
            scatter_kws={'marker':"o", 'edgecolors':'black'},
             fill_color='teal')
```



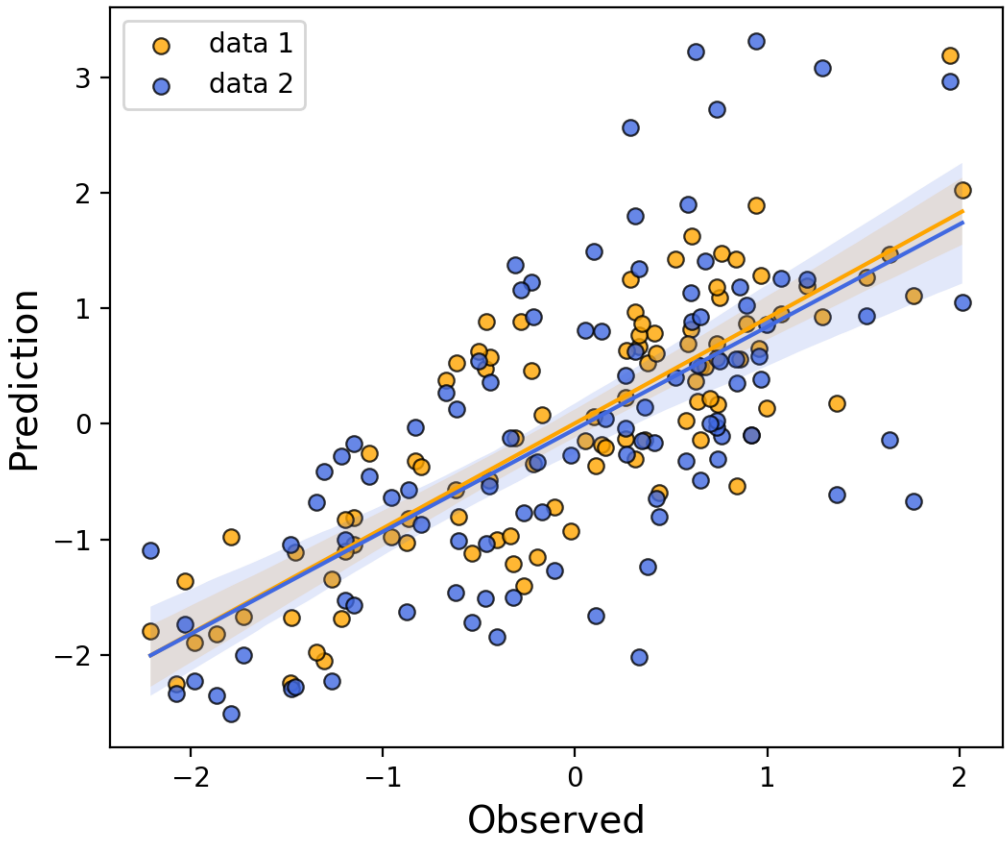
hiding confidence interval

```
_ = regplot(x, y, marker_color='crimson', marker_size=40,
            scatter_kws={'marker':"o", 'edgecolors':'black'},
             ci=None, line_color='olive')
```



multiple regression lines with customized marker, line and fill style

```
cov = np.array(
    [[1.0, 0.9, 0.7],
     [0.9, 1.2, 0.8],
     [0.7, 0.8, 1.4]]
)
data = rng.multivariate_normal(np.zeros(3),
                               cov, size=100)

ax = regplot(data[:, 0], data[:, 1], line_color='orange',
             marker_color='orange', marker_size=35, fill_color='orange',
             scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
             show=False, label="data 1")
_ = regplot(data[:, 0], data[:, 2], line_color='royalblue', ax=ax,
                marker_color='royalblue', marker_size=35, fill_color='royalblue',
             scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
             show=False, label="data 2", ax_kws=dict(legend_kws=dict(loc=(0.1, 0.8))))
plt.show()
```



Total running time of the script: (0 minutes 2.345 seconds)

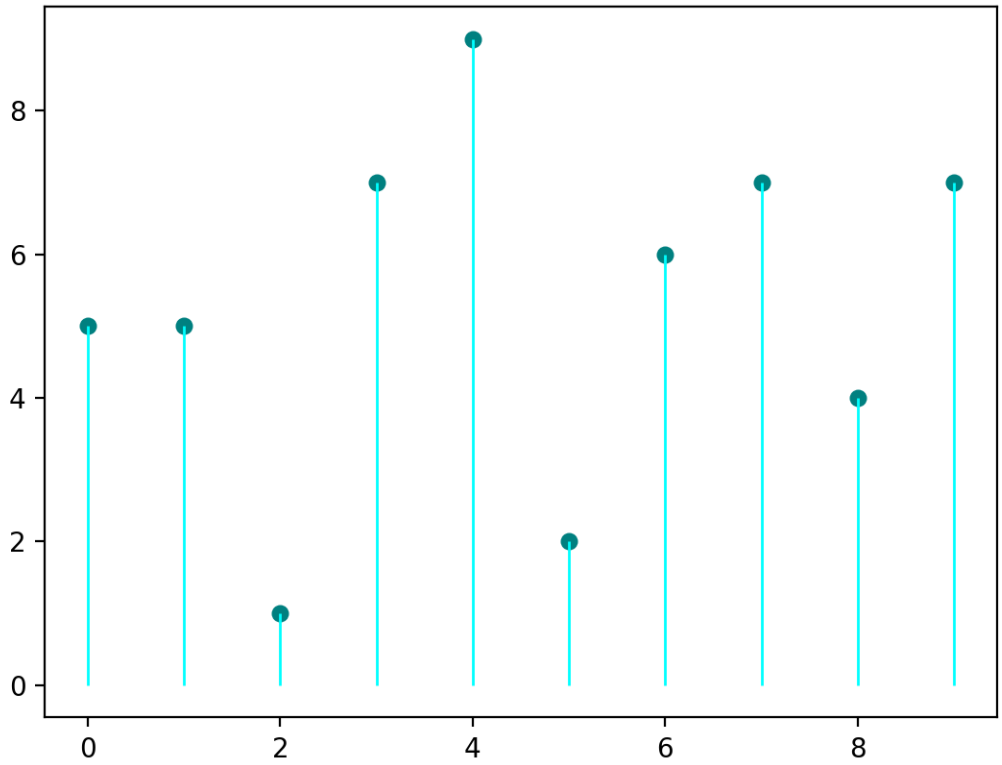
6.6 lollipop plot

```
# sphinx_gallery_thumbnail_number = -1

import matplotlib.pyplot as plt
import numpy as np
from easy_mpl import lollipop_plot
from easy_mpl.utils import version_info

version_info()
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

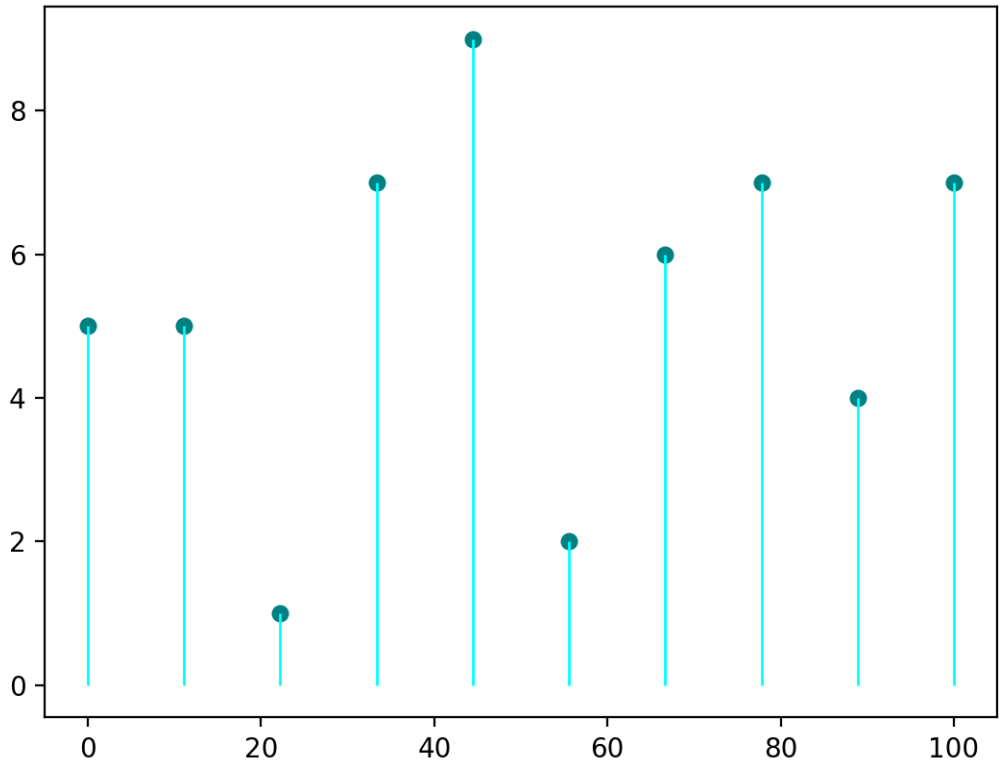
To draw a lollipop we need an array or a list of numeric values

```
y = np.random.randint(1, 10, size=10)

_ = lollipop_plot(y, title="vanilla")
```



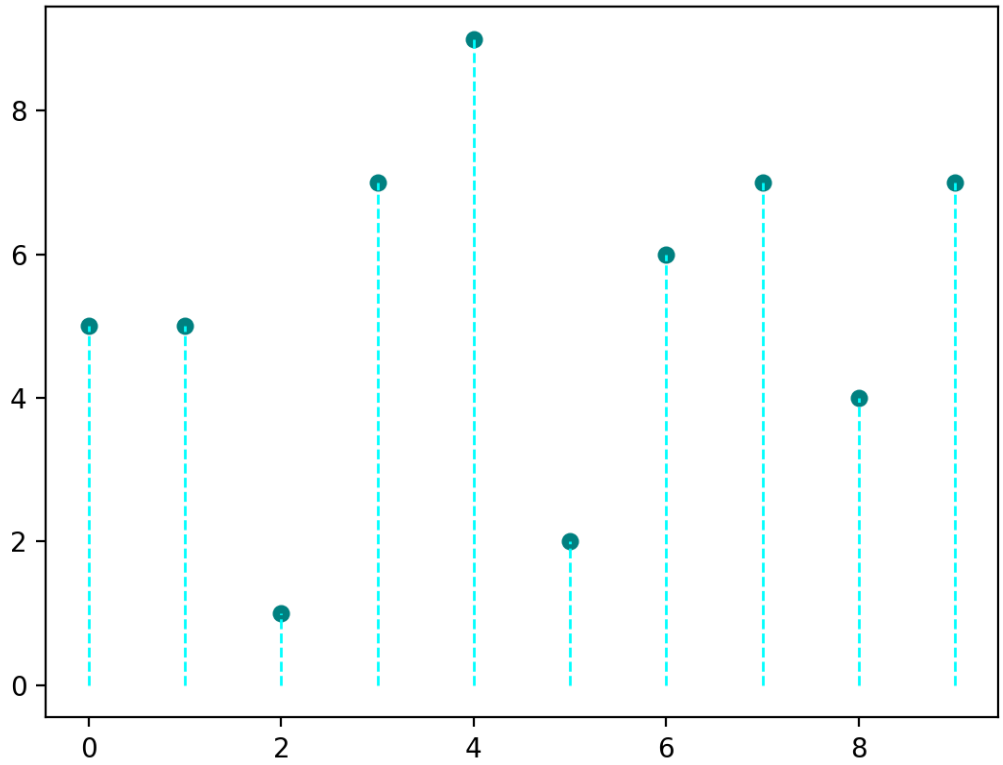
We can also specify the x coordinates for our data as second argument

```
_ = lollipop_plot(y, np.linspace(0, 100, len(y)), title="with x and y")
```



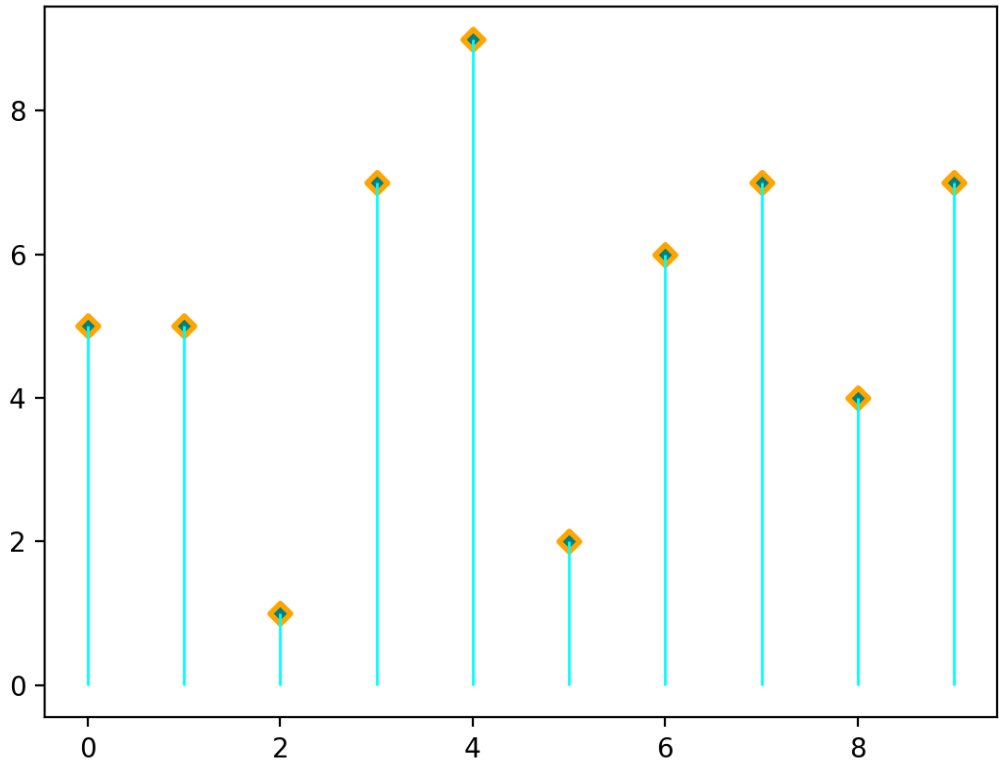
line style can be set using `line_style` argument.

```
_ = lollipop_plot(y, line_style='--', title="with custom linestyle")
```



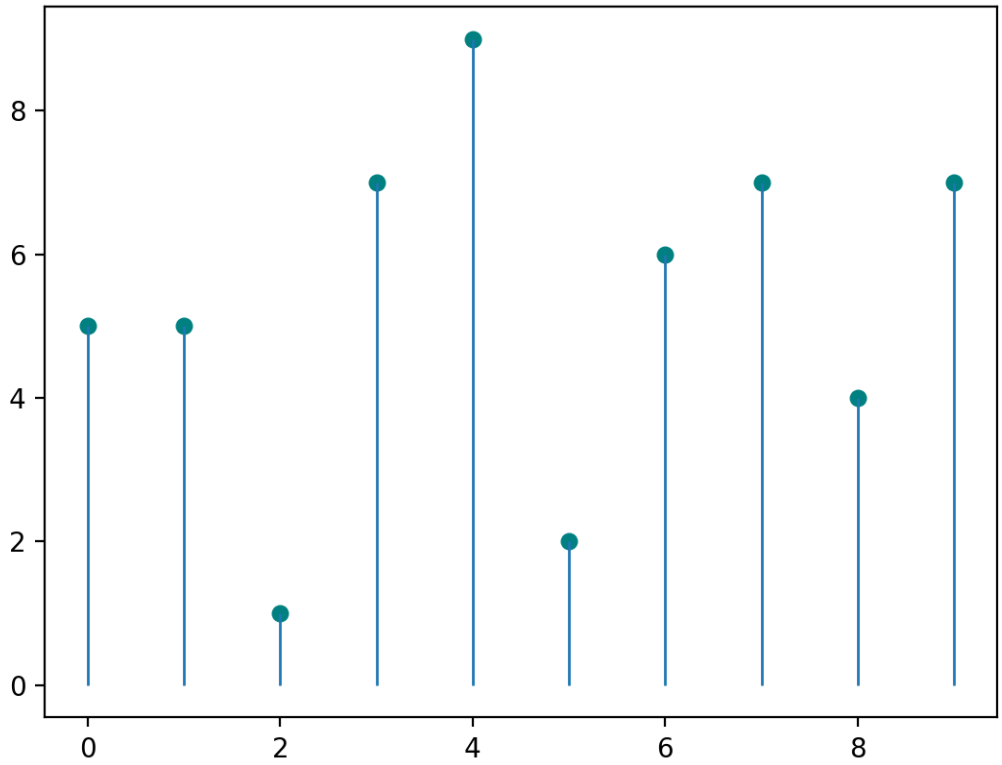
Similarly marker style can be set using `marker_style` argument.

```
_ = lollipop_plot(y, marker_style='D',
                  marker_kws=dict(edgecolor="orange", linewidth=2))
```



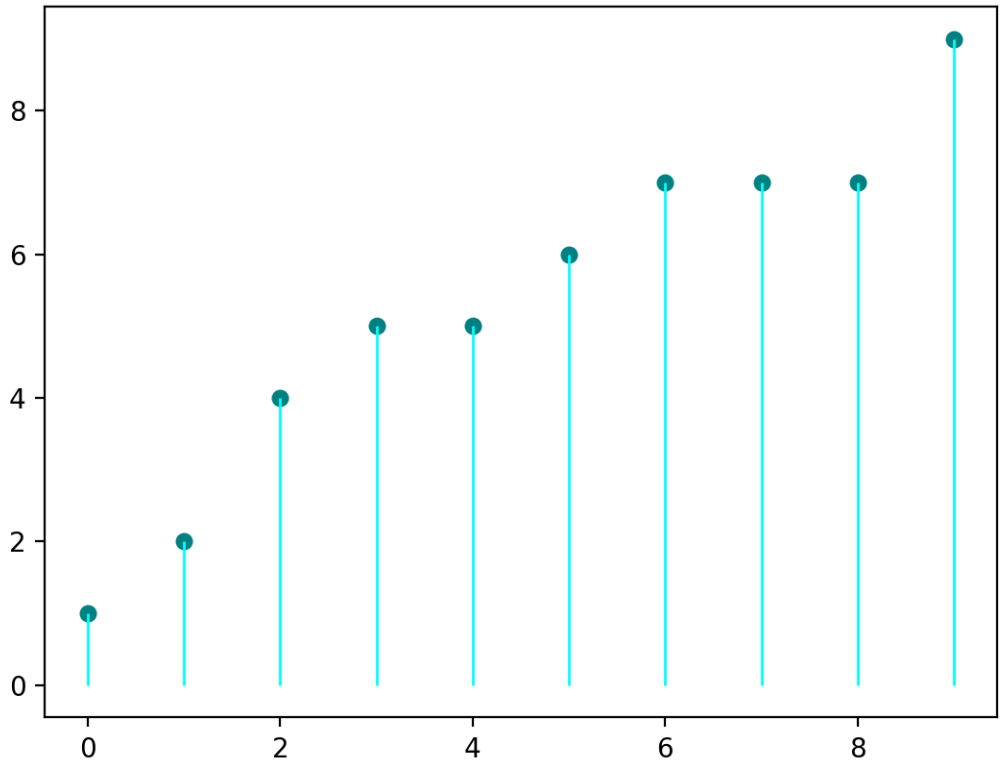
the line color can also be a matplotlib colormap name

```
_ = lollipop_plot(y, line_color="RdBu")
```



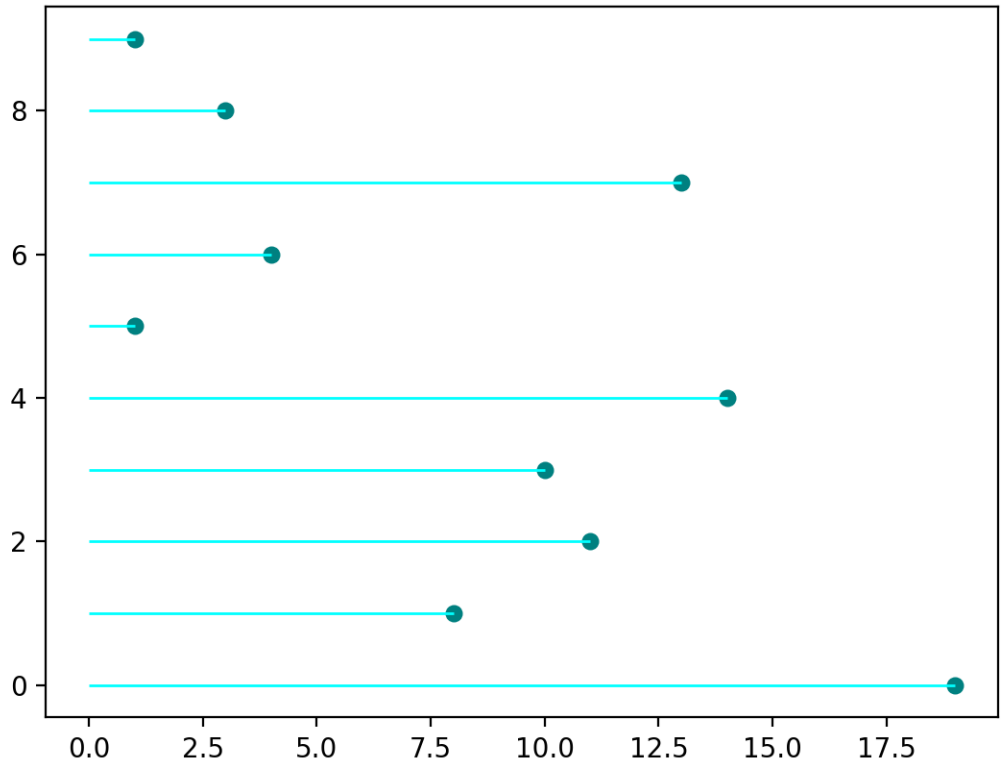
We can sort the lollipops by setting the `sort` to True

```
_ = lollipop_plot(y, sort=True, title="sort")
```



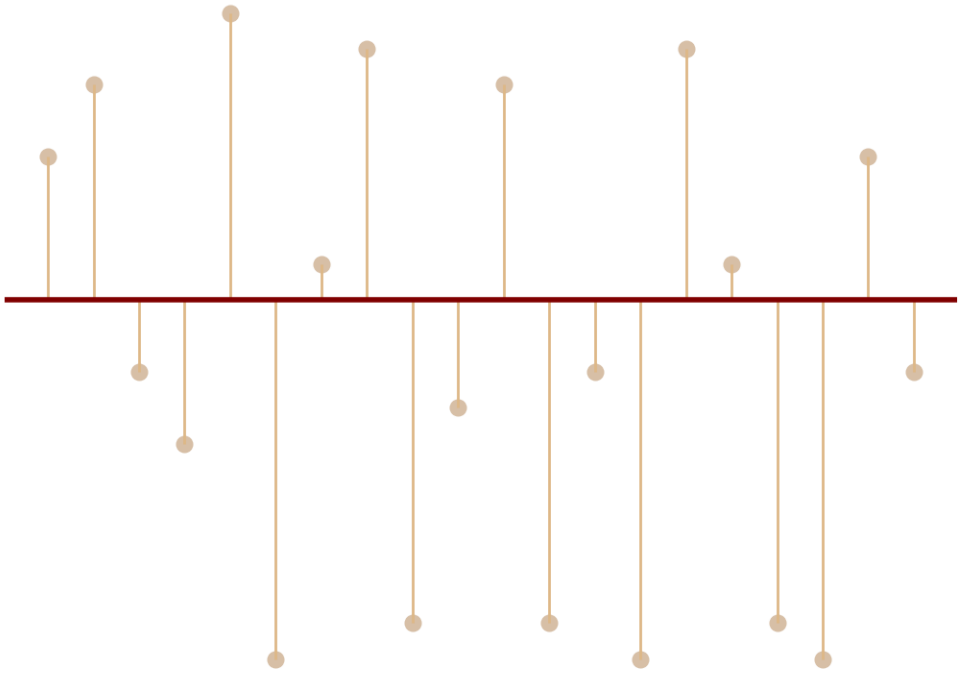
The orientation of lollipops can be made horizontal

```
y = np.random.randint(0, 20, size=10)
_ = lollipop_plot(y, orientation="horizontal", title="horizontal")
```



The lollipop plot returns matplotlib axes object which can be used for further manipulation of axes.

```
y = np.random.randint(-10, 10, 20)
y[y==0] = 1
ax = lollipop_plot(y, marker_color="#D7BFA6",
                   line_color="burlywood",
                   show=False)
ax.axhline(0.0, lw=2.0, color='maroon')
ax.axis('off')
plt.show()
```



Total running time of the script: (0 minutes 1.632 seconds)

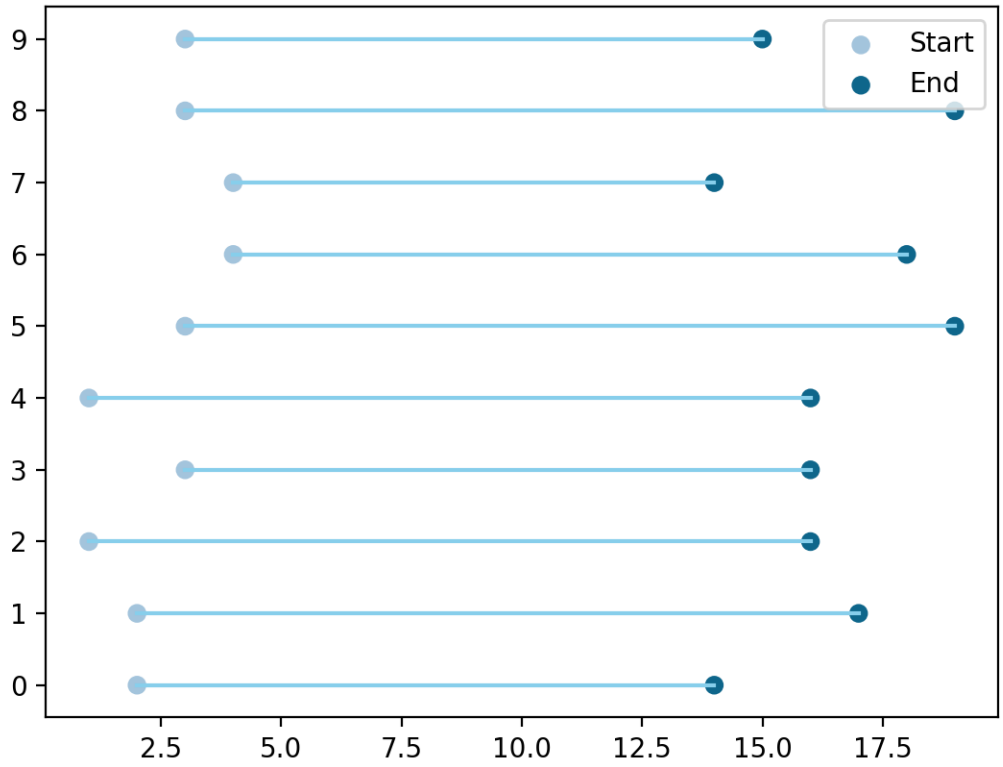
6.7 dumbell plot

```
# sphinx_gallery_thumbnail_number = -2

import matplotlib.pyplot as plt
import numpy as np
from easy_mpl import dumbbell_plot
from easy_mpl.utils import version_info
from easy_mpl.utils import despine_axes

version_info()
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

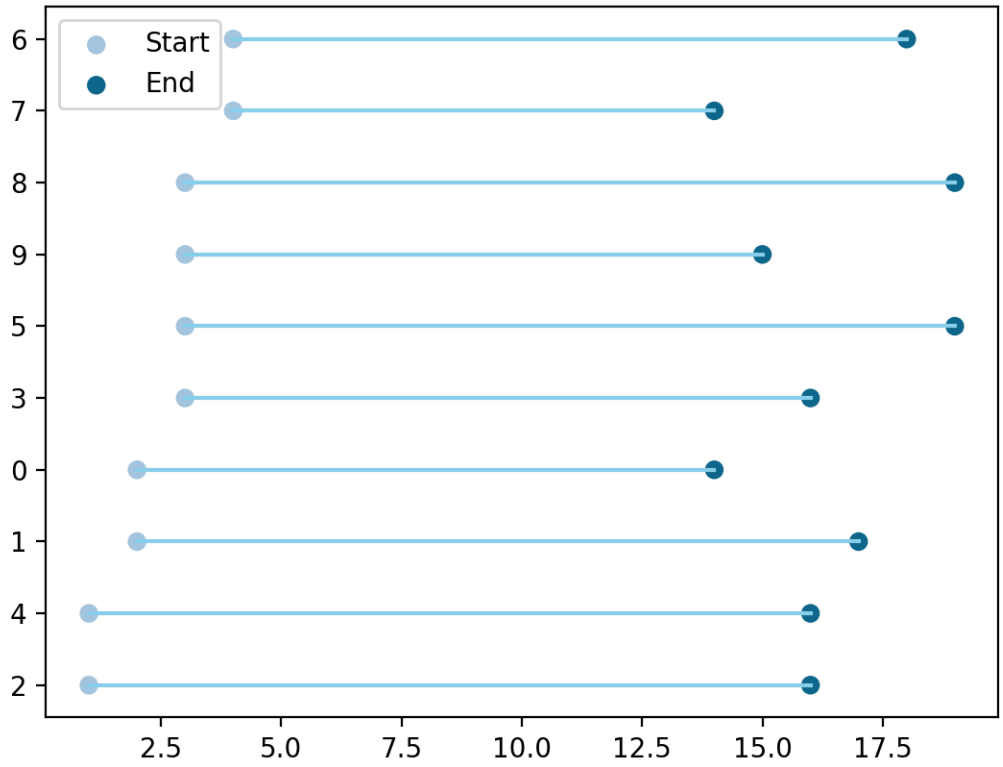
To plot a dumbbell, we require two arrays of equal length.

```
st = np.random.randint(1, 5, 10)
en = np.random.randint(11, 20, 10)

_ = dumbbell_plot(st, en)
```



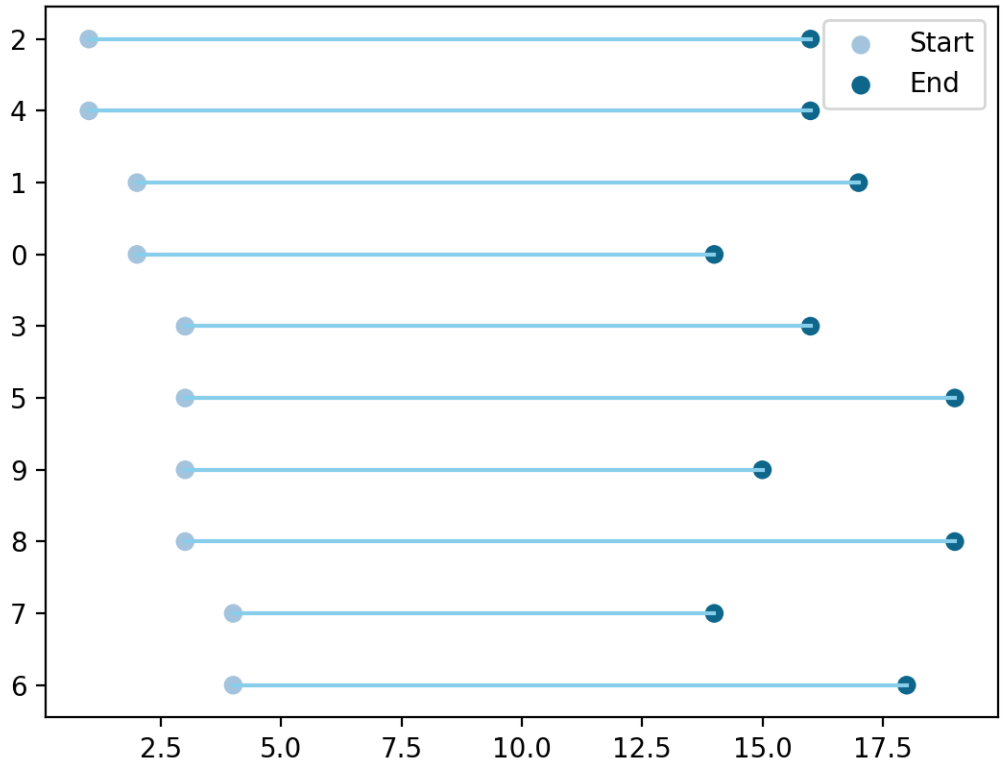
We can sort the dumbbells according the starting value

```
_ = dumbbell_plot(st, en, sort_start="ascend")
```



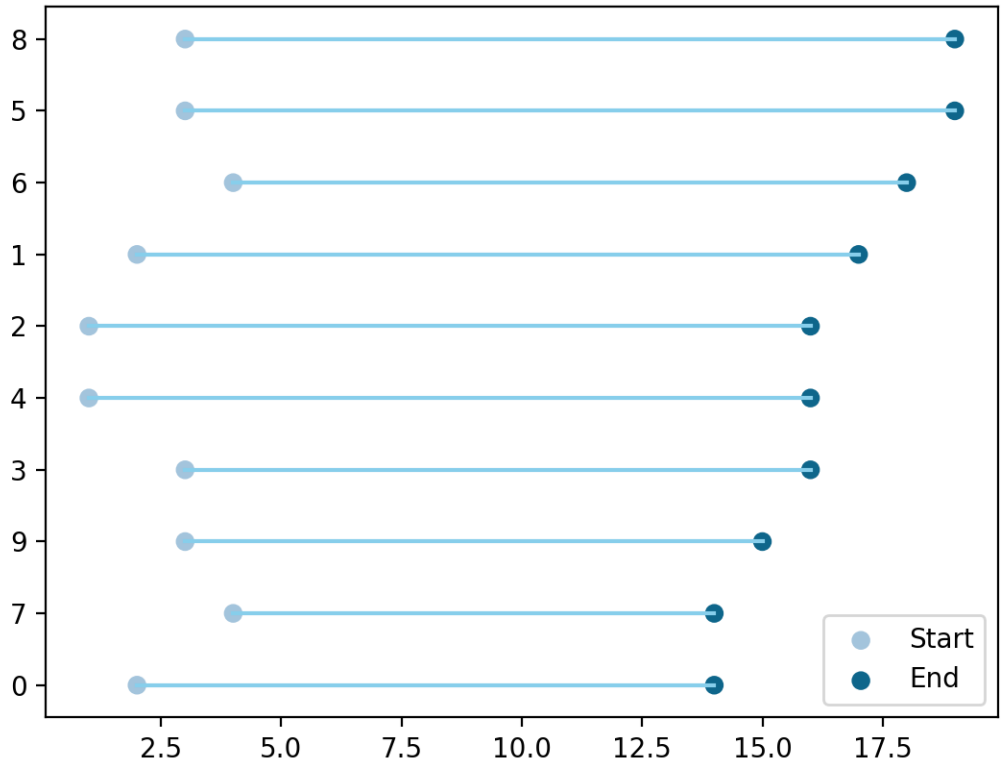
The sorting can also be in descending order

```
_ = dumbbell_plot(st, en, sort_start="descend")
```



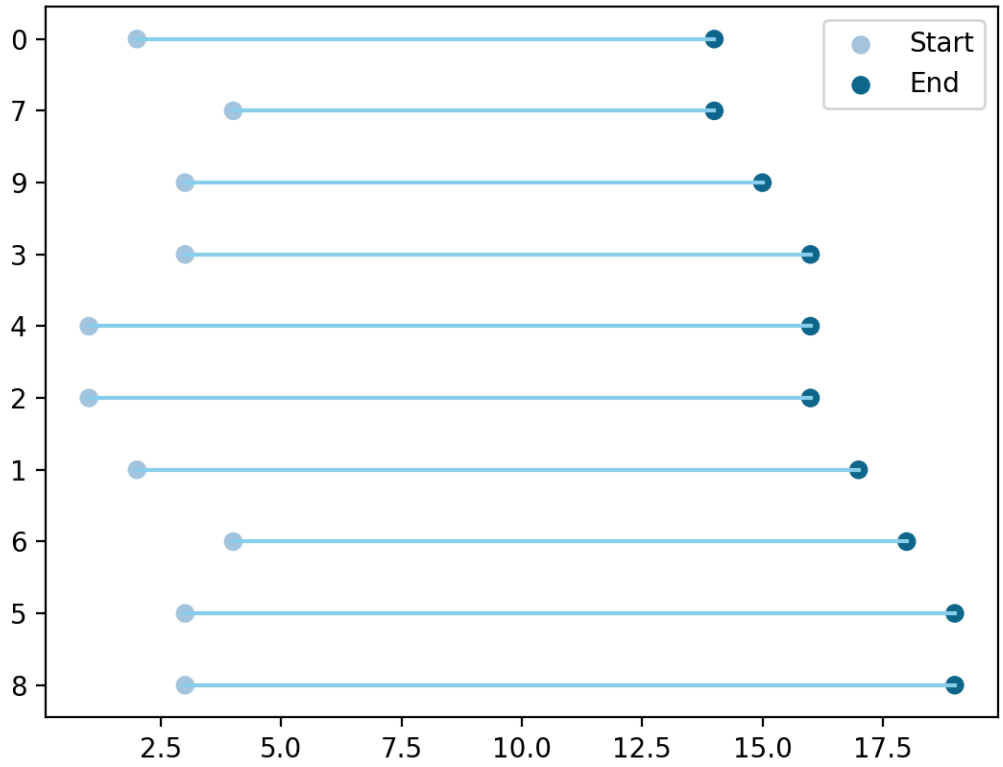
We can also sort dumbbells according to end value

```
_ = dumbbell_plot(st, en, sort_end="ascend")
```

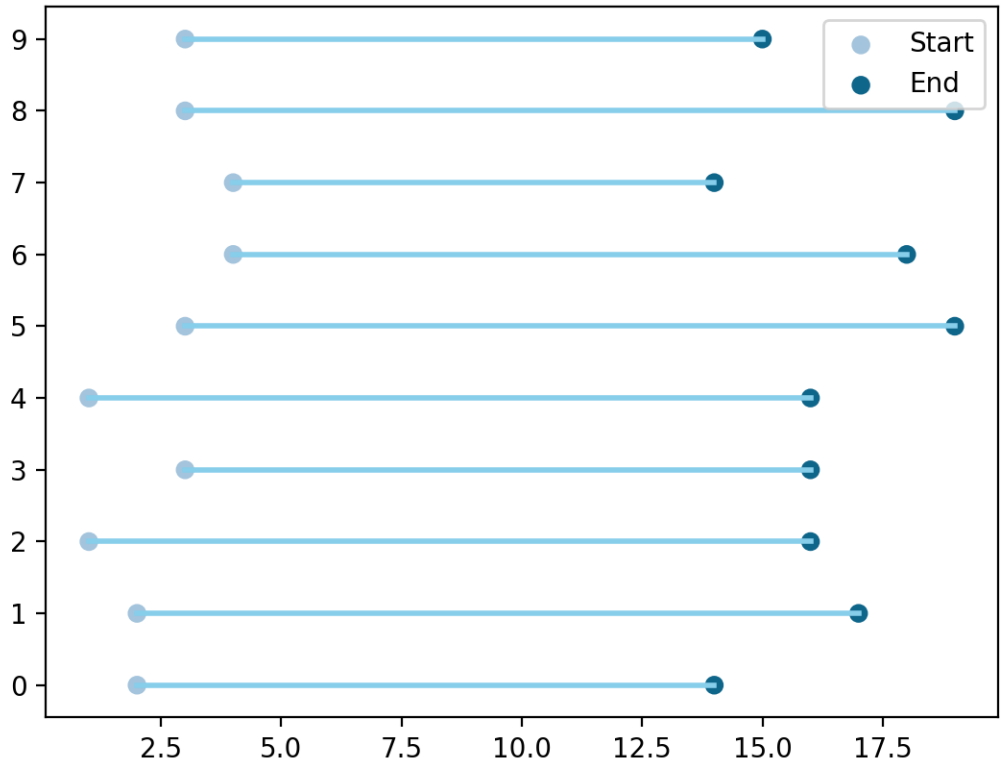


And this can also be in descending order

```
_ = dumbbell_plot(st, en, sort_end="descend")
```

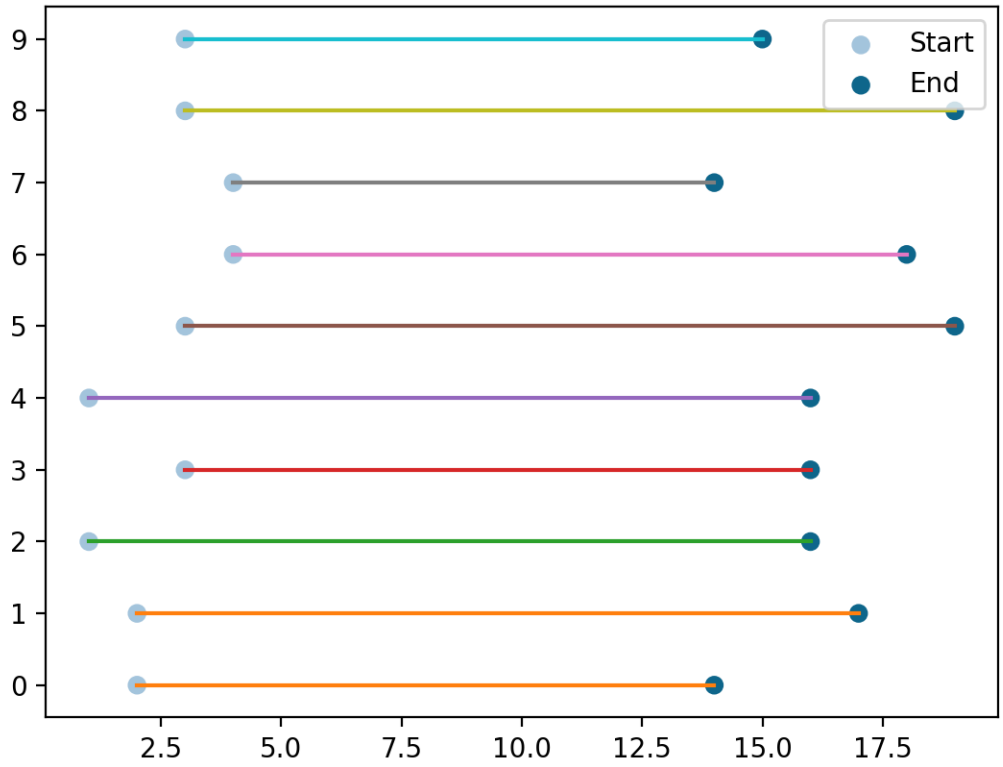



```
# The properties of line can be modified by providing ``line_kws`` dictionary.
_ = dumbbell_plot(st, en, line_kws={'lw':"2"})
```



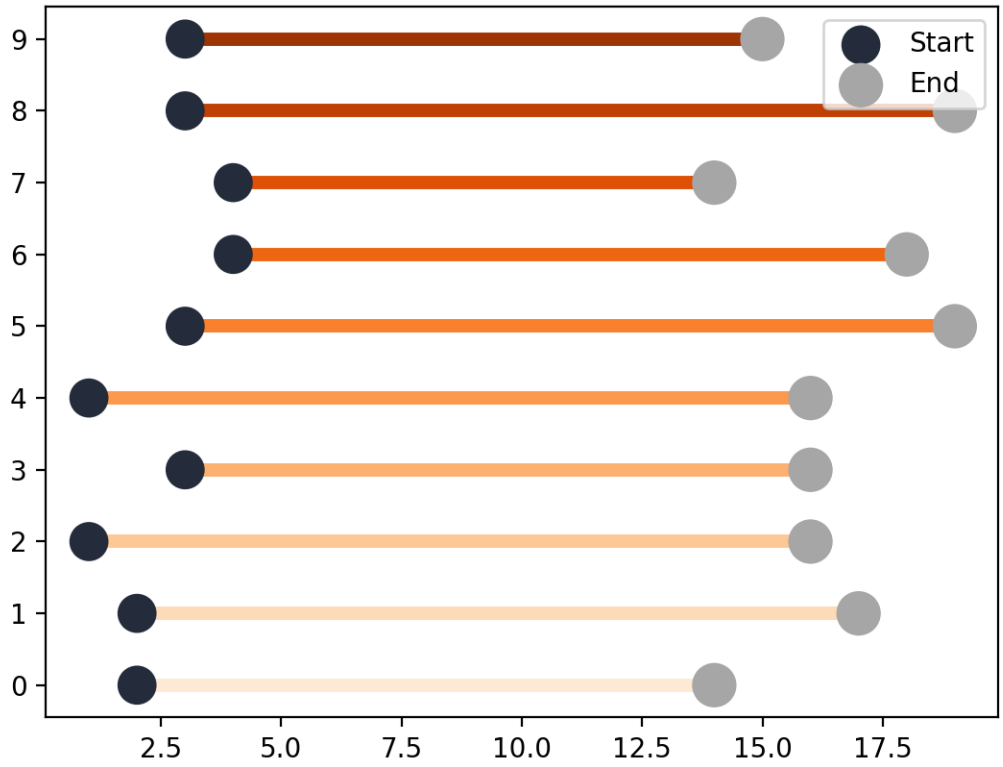
We can also specify the line color using a color palette (color map).

```
_ = dumbbell_plot(st, en, line_color='tab10')
```



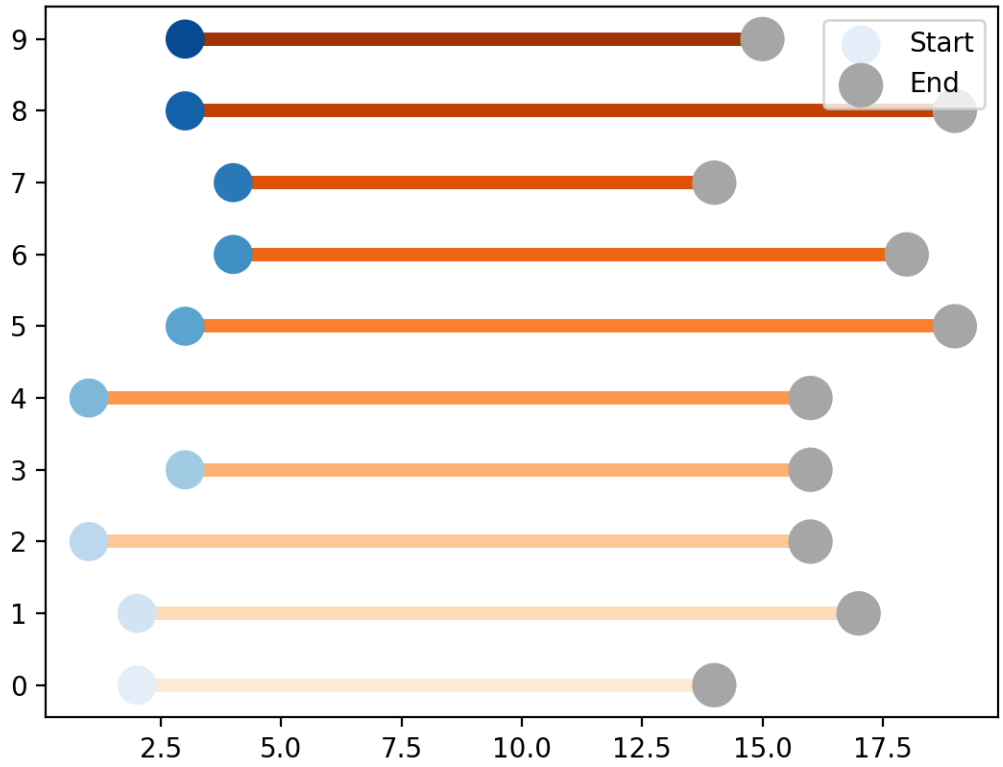
The properties of starting and end markers can be modified by making use of `start_kws` and `end_kws` keyword.

```
_ = dumbbell_plot(
    st, en,
    line_color="Oranges",
    line_kws=dict(lw=5),
    start_kws=dict(s=160, lw=2, zorder=2, color="#242c3c", edgecolors="#242c3c"),
    end_kws=dict(s=200, color="#a6a6a6", edgecolors="#a6a6a6", lw=2.5, zorder=2),

)
```



We can also specifiy a separate color for starting markers. One way of doing this is by specifying a palette (colormap) name.

```
_ = dumbbell_plot(
    st, en,
    line_color="Oranges",
    start_marker_color="Blues",
    line_kws=dict(lw=5),
    start_kws=dict(s=160, lw=2, zorder=2),
    end_kws=dict(s=200, color="#a6a6a6", edgecolors="#a6a6a6", lw=2.5, zorder=2)
)
```



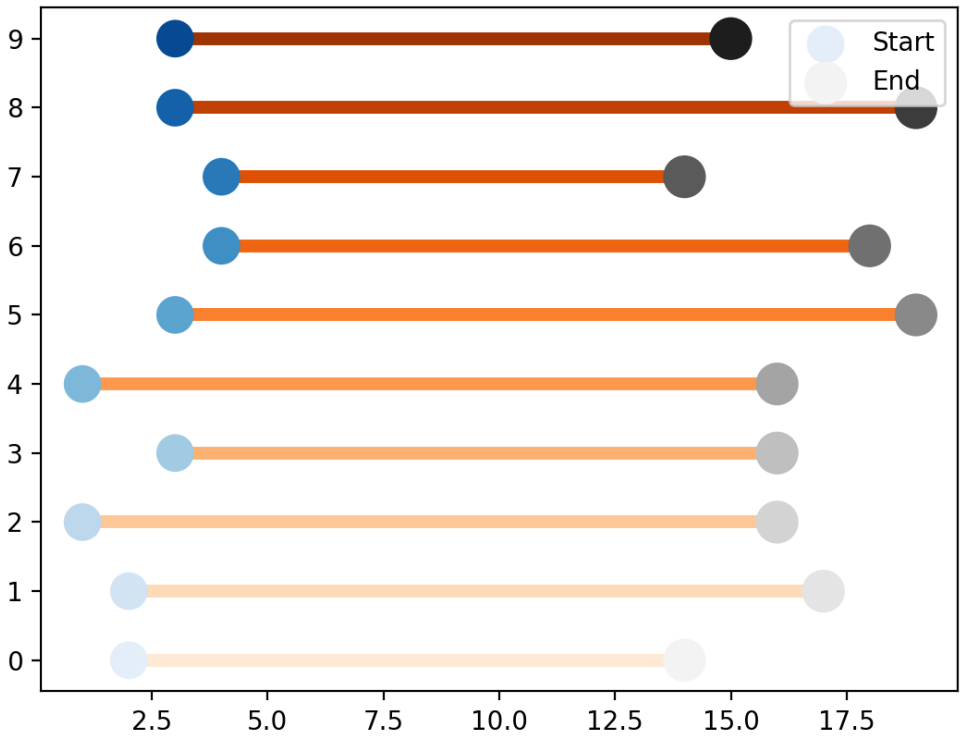
We can also provide colormap for end markers.

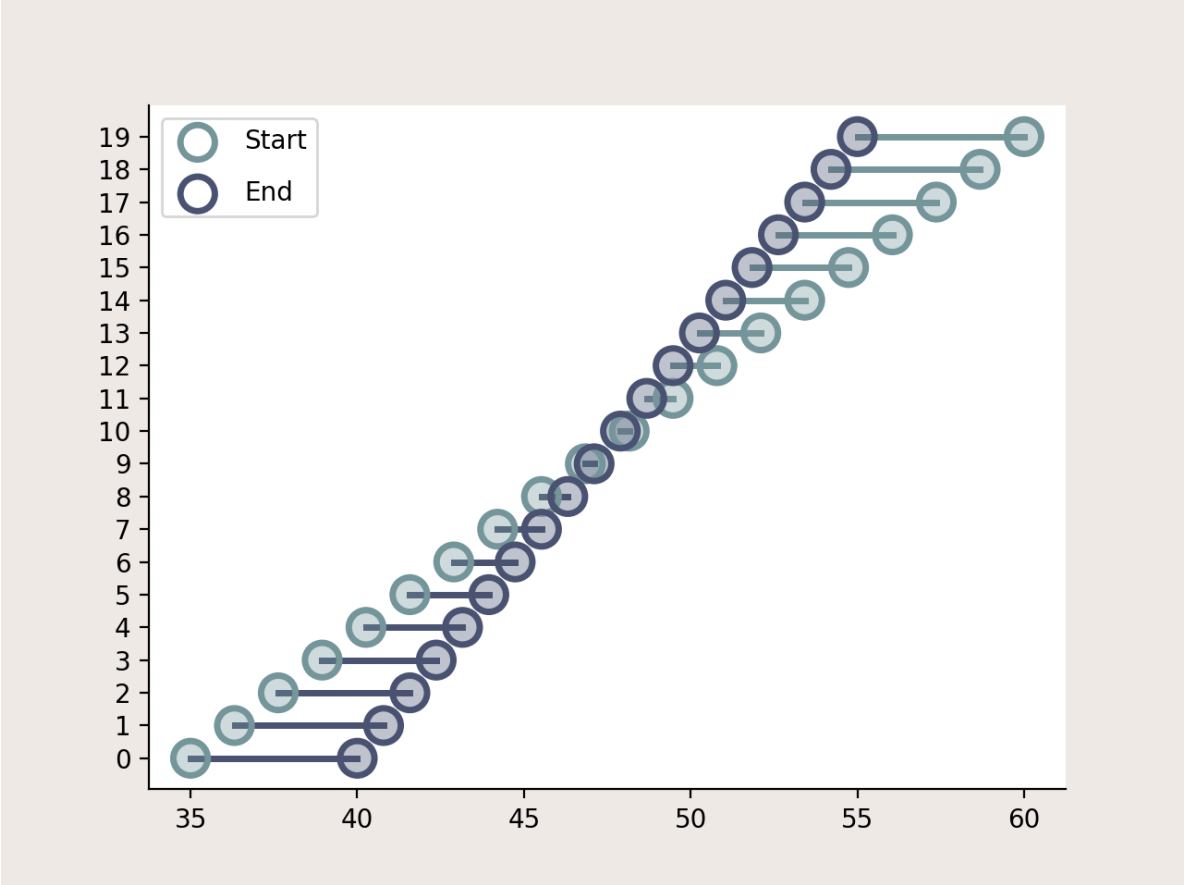
```
_ = dumbbell_plot(
    st, en,
    line_color="Oranges",
    start_marker_color="Blues",
    end_marker_color="Greys",
    line_kws=dict(lw=5),
    start_kws=dict(s=160, lw=2, zorder=2),
    end_kws=dict(s=200, lw=2.5, zorder=2)
)

f, ax = plt.subplots(facecolor = "#EFE9E6")
start = np.linspace(35, 60, 20)
end = np.linspace(40, 55, 20)
line_colors = []
for st, en in zip(start, end):
    if st>en:
        line_colors.append("#74959A")
    else:
        line_colors.append("#495371")

dumbbell_plot(start, end,
              start_kws=dict(color = "#74959A", s = 150, alpha = 0.35, zorder = 3),
```

(continues on next page)

(continued from previous page)

```
                end_kws=dict(color = "#495371", s = 150, alpha = 0.35, zorder = 3),
                line_kws=dict(zorder = 2, lw = 2.5), line_color=line_colors,
                ax=ax,
                show=False, )

dumbbell_plot(start, end,
                start_kws=dict(color = "none", ec = "#74959A", s = 180, lw = 2.5, zorder =␣
↪3),
                end_kws=dict(color = "none", ec = "#495371", s = 180, lw = 2.5, zorder =␣
↪3),
                line_kws=dict(zorder = 2, lw = 2.5), line_color=line_colors,
                ax=ax,
                show=False )
despine_axes(ax, keep=['left', 'bottom'])
lines, labels = ax.get_legend_handles_labels()
ax.legend([lines[2], lines[3]], ['Start', 'End'], labelspacing=1.0)
plt.show()
```





Total running time of the script: (0 minutes 2.981 seconds)

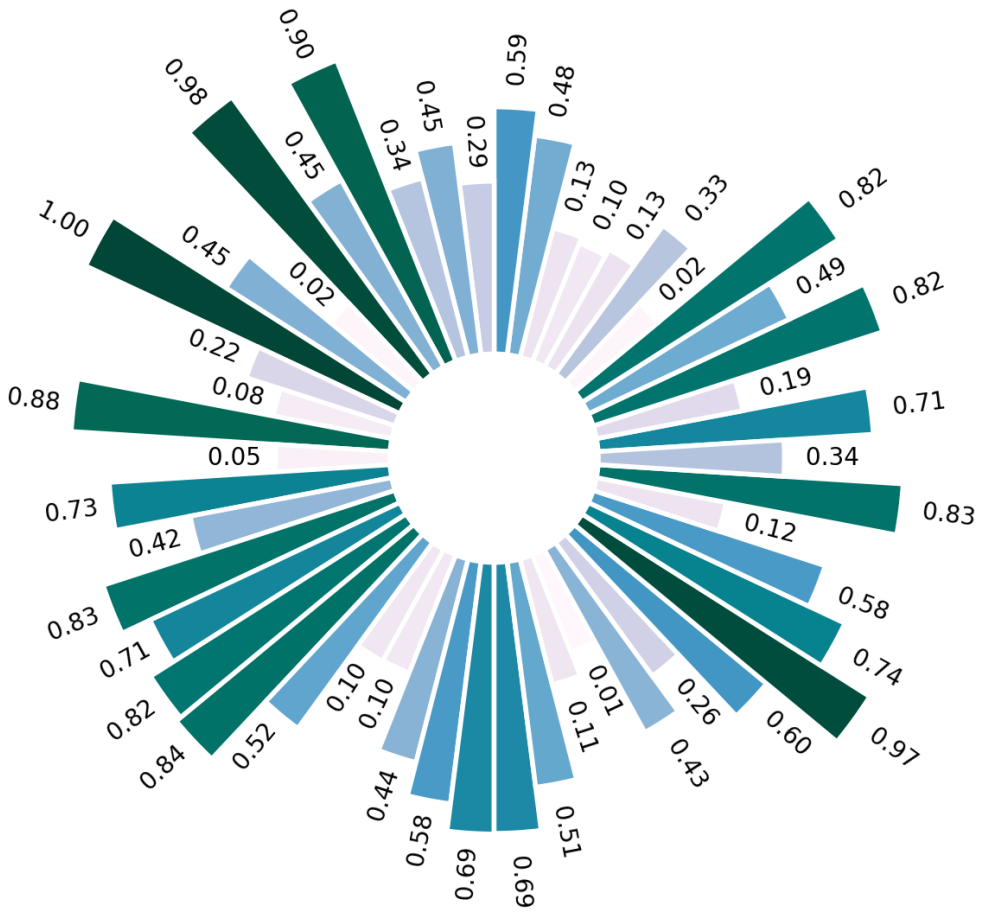
6.8 circular_bar plot

```
import numpy as np
from easy_mpl import circular_bar_plot
from easy_mpl.utils import version_info

version_info()
```

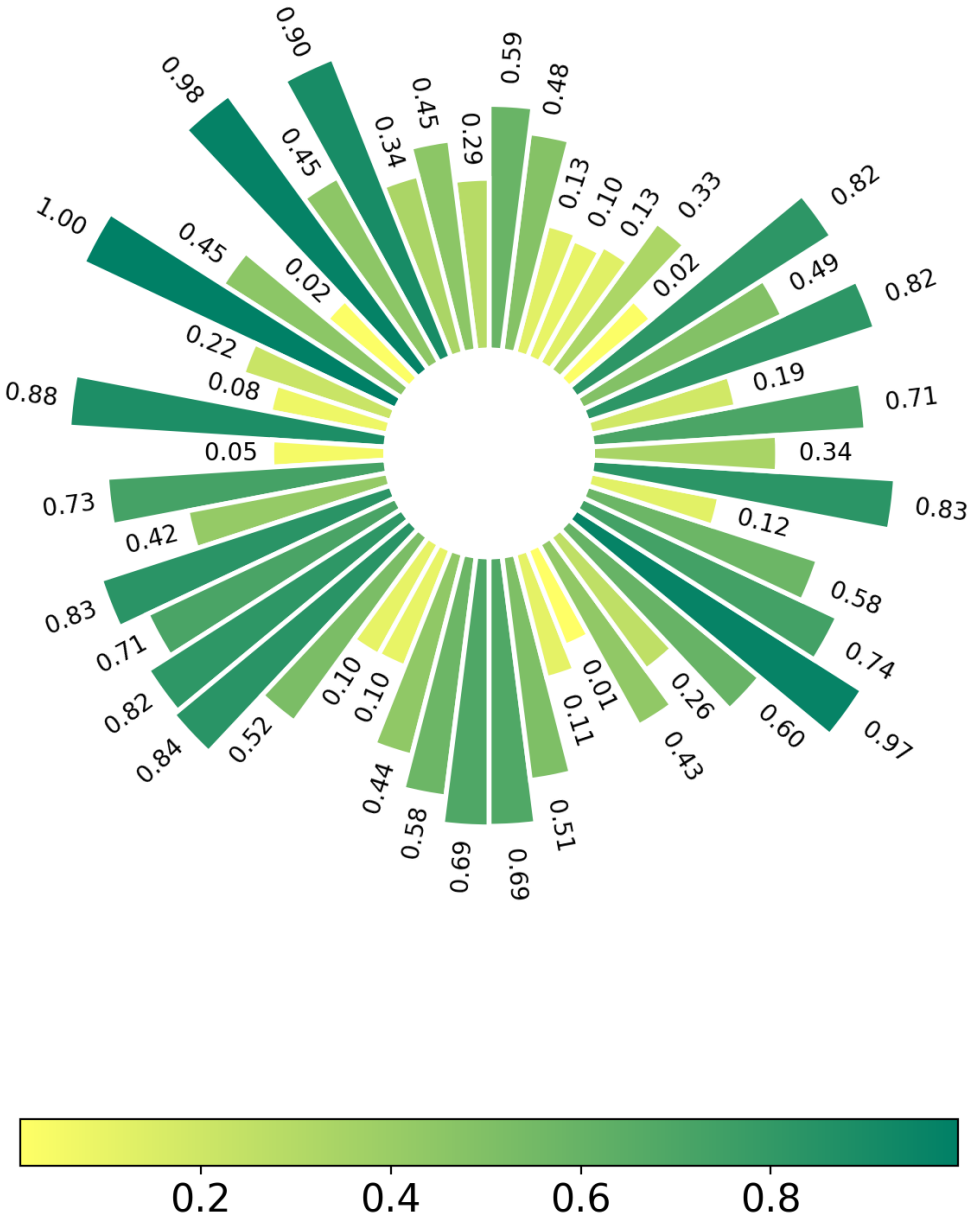
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

basic

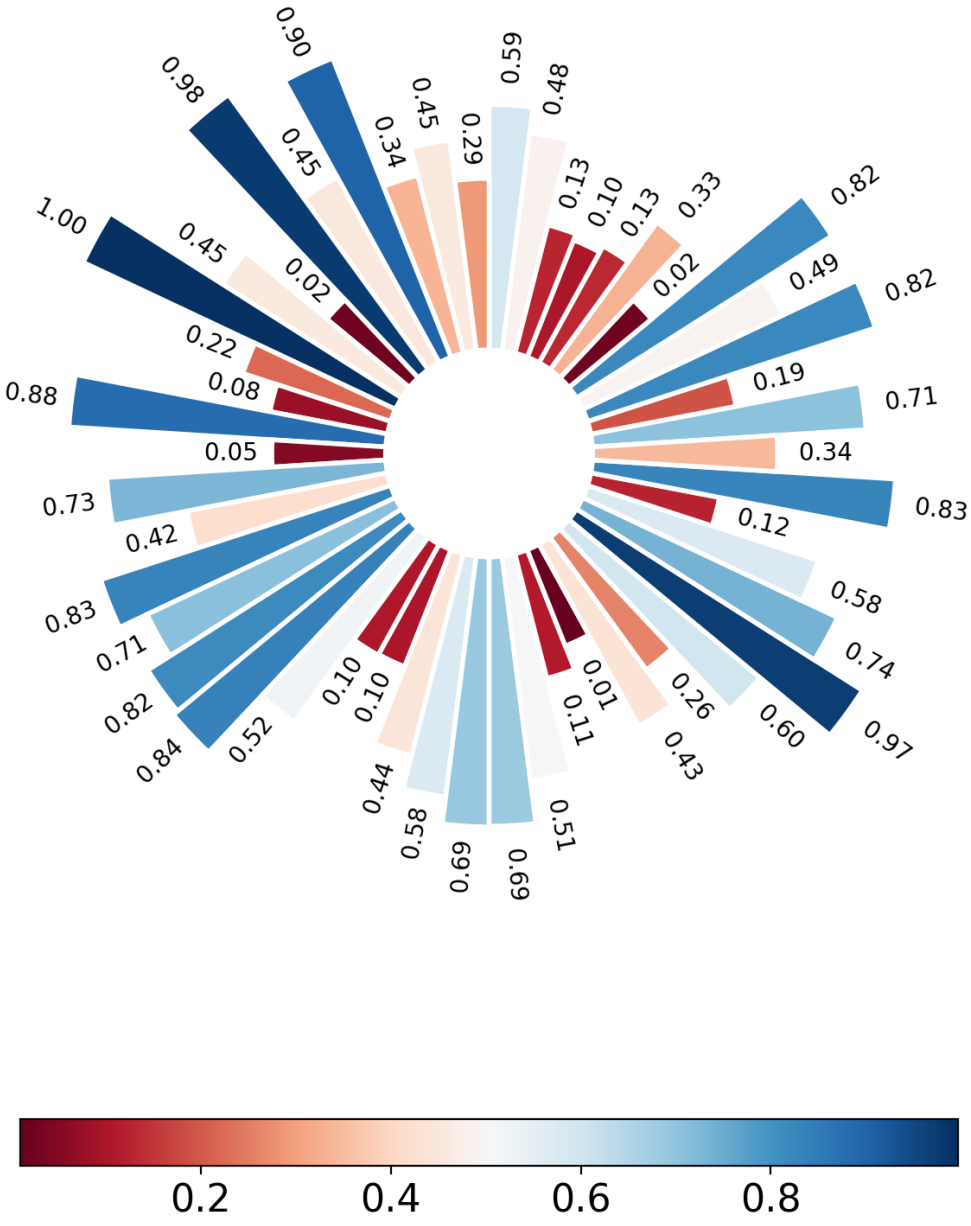
```
data = np.random.random(50, )

_ = circular_bar_plot(data)
```


0.59
0.48
0.13
0.10
0.13
0.33
0.82
0.49
0.82
0.19
0.71
0.34
0.83
0.12
0.58
0.74
0.97
0.60
0.26
0.43
0.01
0.11
0.51
0.69
0.69
0.58
0.44
0.10
0.10
0.52
0.84
0.82
0.71
0.83
0.42
0.73
0.05
0.88
0.08
0.22
1.00
0.45
0.02
0.45
0.98
0.90
0.34
0.45
0.29

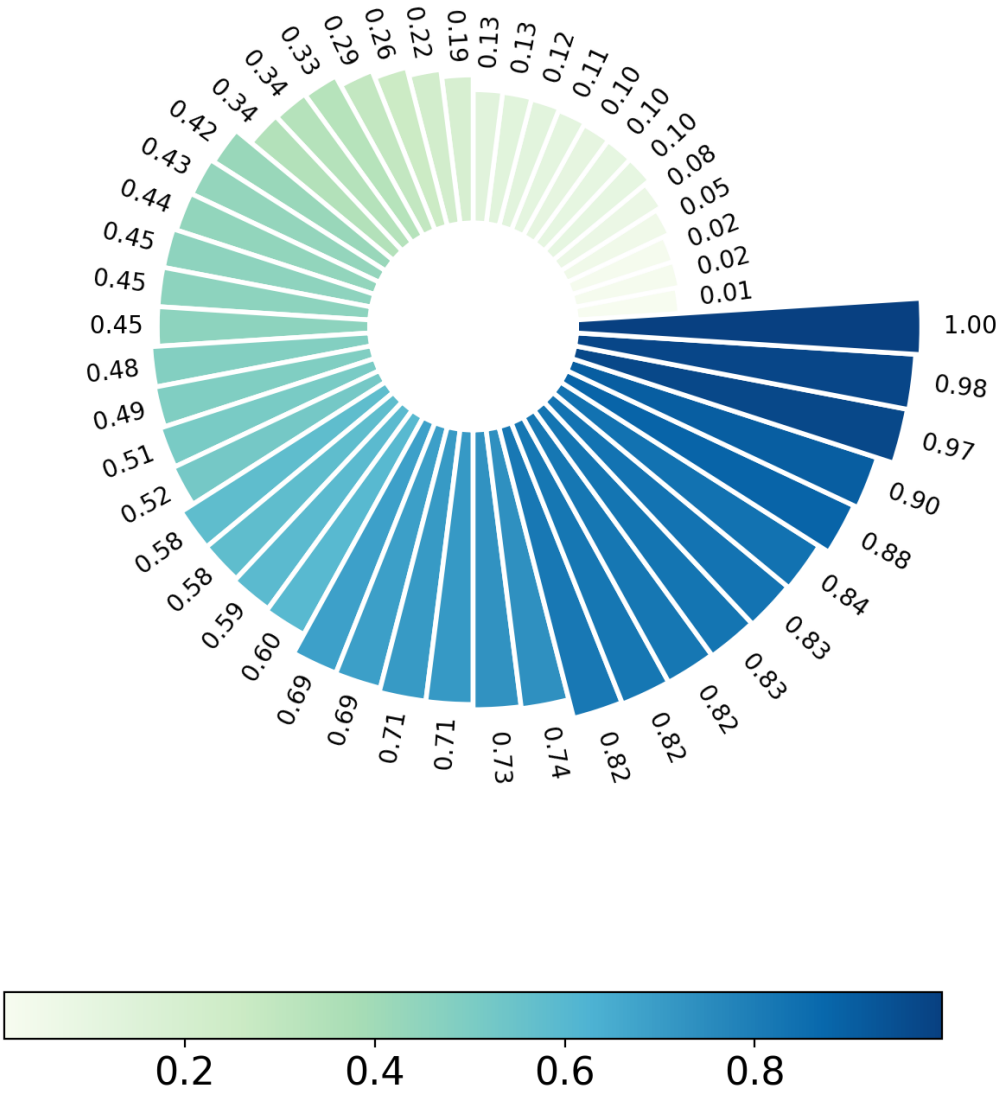
```
_ = circular_bar_plot(data, colorbar=True)
```


0.59
0.48
0.13
0.10
0.13
0.33
0.82
0.49
0.82
0.19
0.71
0.34
0.83
0.12
0.58
0.74
0.97
0.60
0.26
0.43
0.01
0.11
0.51
0.69
0.69
0.58
0.44
0.10
0.10
0.52
0.84
0.82
0.71
0.83
0.42
0.73
0.05
0.88
0.08
0.22
0.02
0.45
1.00
0.45
0.98
0.90
0.34
0.45
0.29
0.2
0.4
0.6
0.8

```
_ = circular_bar_plot(data, color="RdBu", colorbar=True)
```

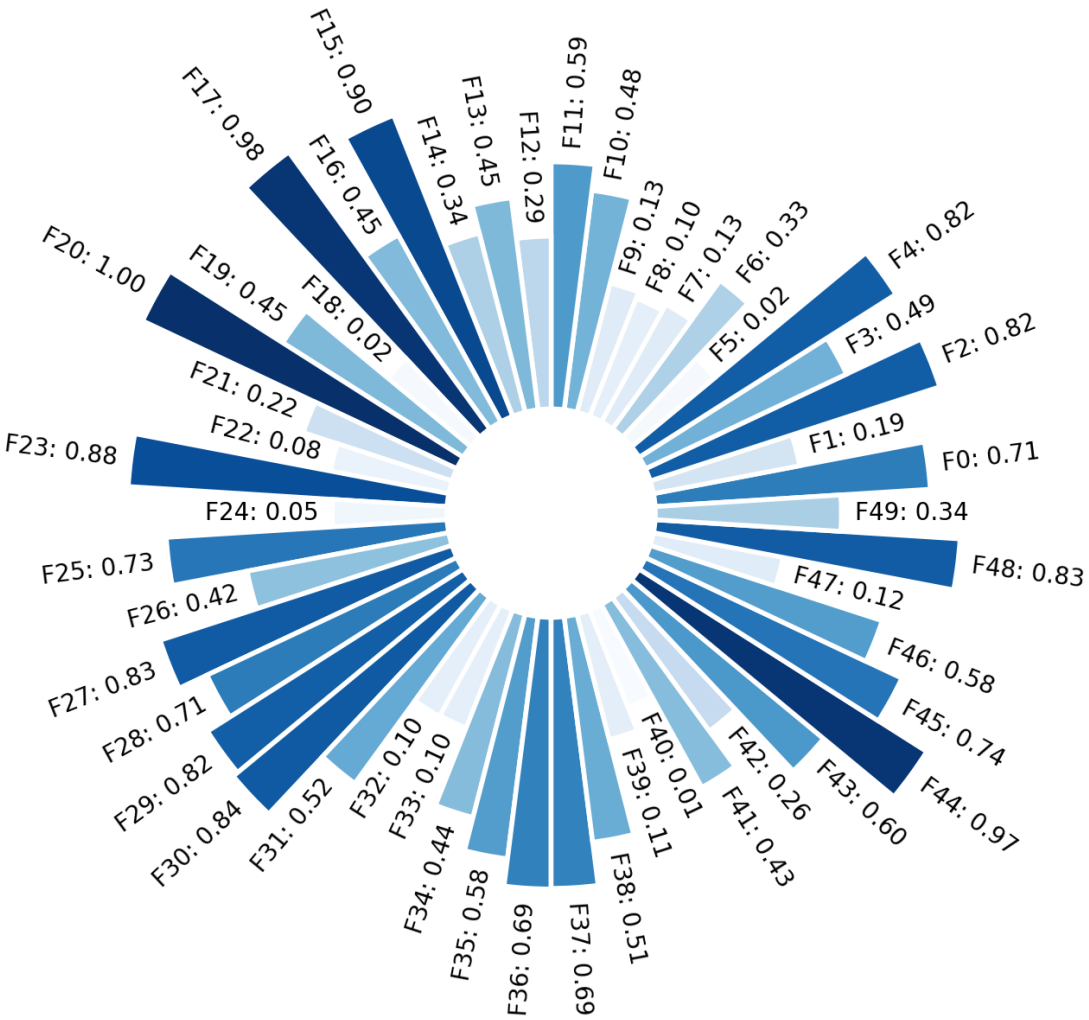

0.59
0.48
0.13
0.10
0.13
0.33
0.82
0.49
0.82
0.19
0.71
0.34
0.83
0.12
0.58
0.74
0.97
0.60
0.43
0.26
0.01
0.11
0.51
0.69
0.69
0.58
0.44
0.10
0.10
0.52
0.84
0.82
0.71
0.83
0.42
0.73
0.05
0.88
0.08
0.22
0.02
0.45
1.00
0.45
0.98
0.90
0.34
0.45
0.29
0.2
0.4
0.6
0.8

```
_ = circular_bar_plot(data, sort=True, colorbar=True)
```


1.00
0.98
0.97
0.90
0.88
0.84
0.83
0.83
0.82
0.82
0.74
0.73
0.71
0.70
0.69
0.60
0.59
0.58
0.58
0.52
0.51
0.49
0.48
0.45
0.45
0.45
0.44
0.43
0.42
0.34
0.34
0.33
0.29
0.26
0.22
0.19
0.13
0.13
0.12
0.11
0.10
0.10
0.10
0.08
0.05
0.02
0.02
0.01
0.2
0.4
0.6
0.8

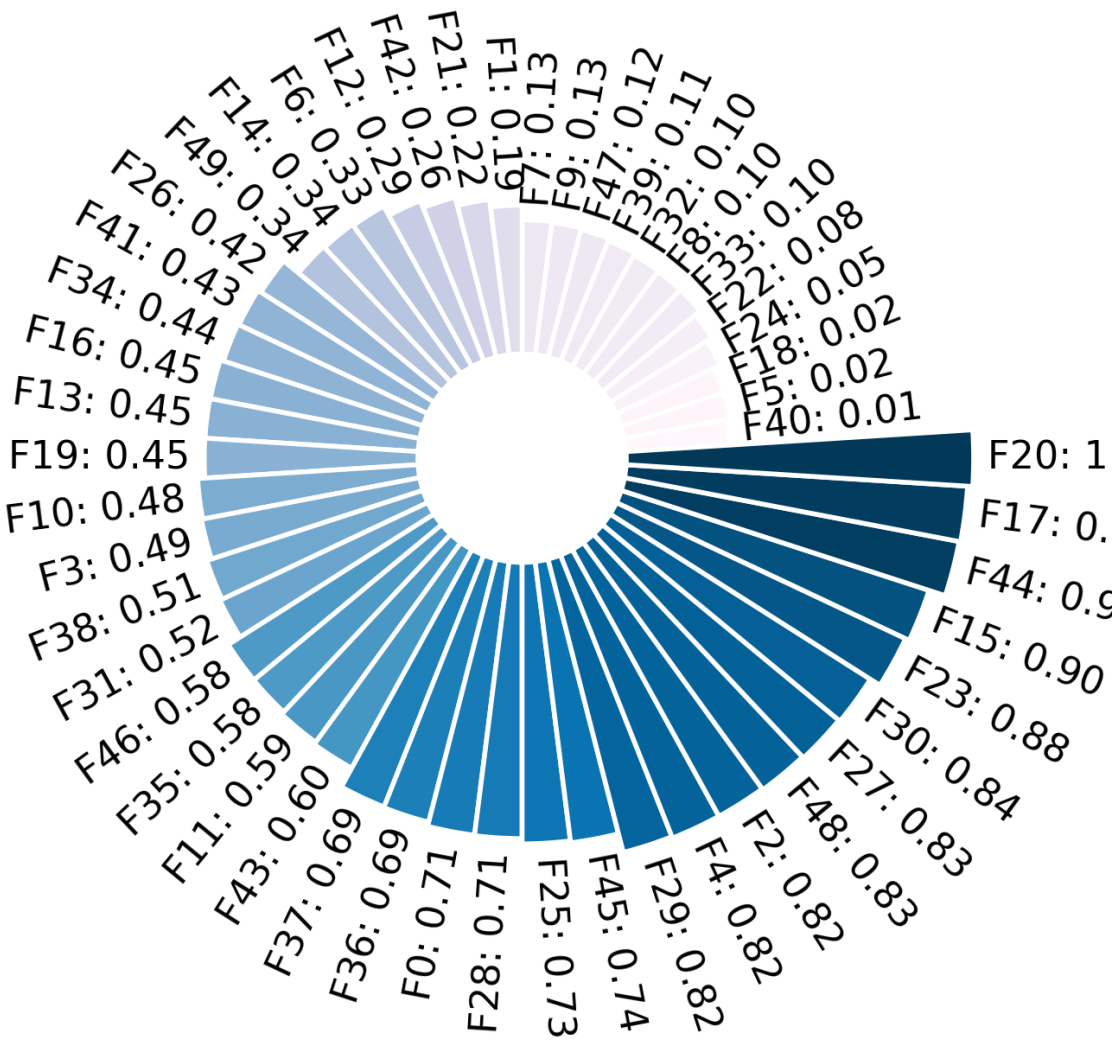
with names

```
names = [f"F{i}" for i in range(len(data))]
_ = circular_bar_plot(data, names)
```


F11: 0.59
F10: 0.48
F9: 0.13
F8: 0.10
F7: 0.13
F6: 0.33
F4: 0.82
F3: 0.49
F2: 0.82
F1: 0.19
F0: 0.71
F49: 0.34
F48: 0.83
F47: 0.12
F46: 0.58
F45: 0.74
F44: 0.97
F43: 0.60
F42: 0.26
F41: 0.43
F40: 0.01
F39: 0.11
F38: 0.51
F37: 0.69
F36: 0.69
F35: 0.58
F34: 0.44
F33: 0.10
F32: 0.10
F31: 0.52
F30: 0.84
F29: 0.82
F28: 0.71
F27: 0.83
F26: 0.42
F25: 0.73
F24: 0.05
F23: 0.88
F22: 0.08
F21: 0.22
F20: 1.00
F19: 0.45
F18: 0.02
F16: 0.45
F17: 0.98
F15: 0.90
F14: 0.34
F13: 0.45
F12: 0.29

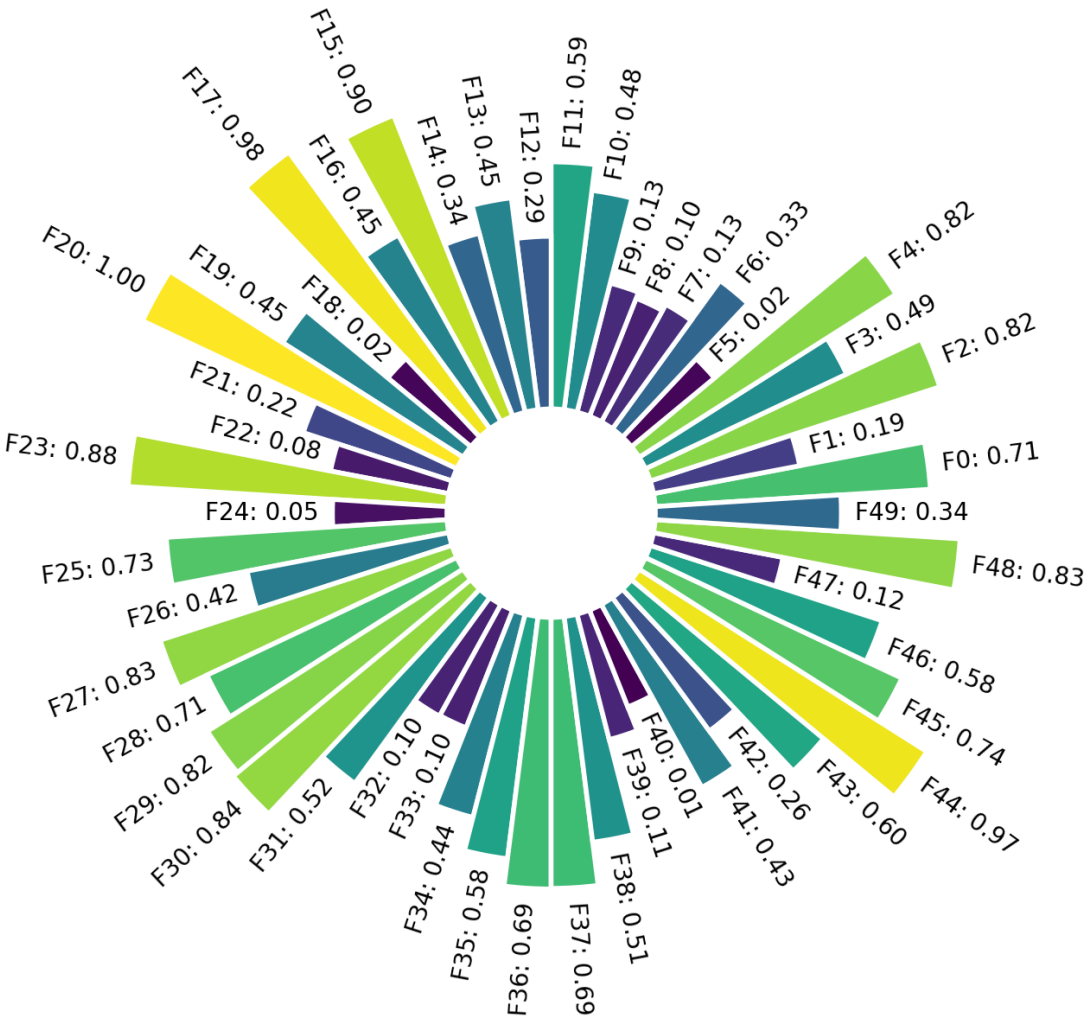
sort values

```
_ = circular_bar_plot(data, names, sort=True, text_kws={"fontsize": 16})
```


F40: 0.01
F5: 0.02
F18: 0.02
F24: 0.05
F22: 0.08
F33: 0.10
F8: 0.10
F32: 0.10
F39: 0.11
F47: 0.12
F9: 0.13
F1: 0.13
F7: 0.19
F21: 0.22
F42: 0.26
F12: 0.29
F6: 0.33
F14: 0.34
F49: 0.34
F26: 0.42
F41: 0.43
F34: 0.44
F16: 0.45
F13: 0.45
F19: 0.45
F10: 0.48
F3: 0.49
F38: 0.51
F31: 0.52
F46: 0.58
F35: 0.58
F11: 0.59
F43: 0.60
F37: 0.69
F36: 0.69
F0: 0.71
F28: 0.71
F25: 0.73
F45: 0.74
F29: 0.82
F4: 0.82
F2: 0.82
F48: 0.83
F27: 0.83
F30: 0.84
F23: 0.88
F15: 0.90
F44: 0.9
F17: 0.
F20: 1

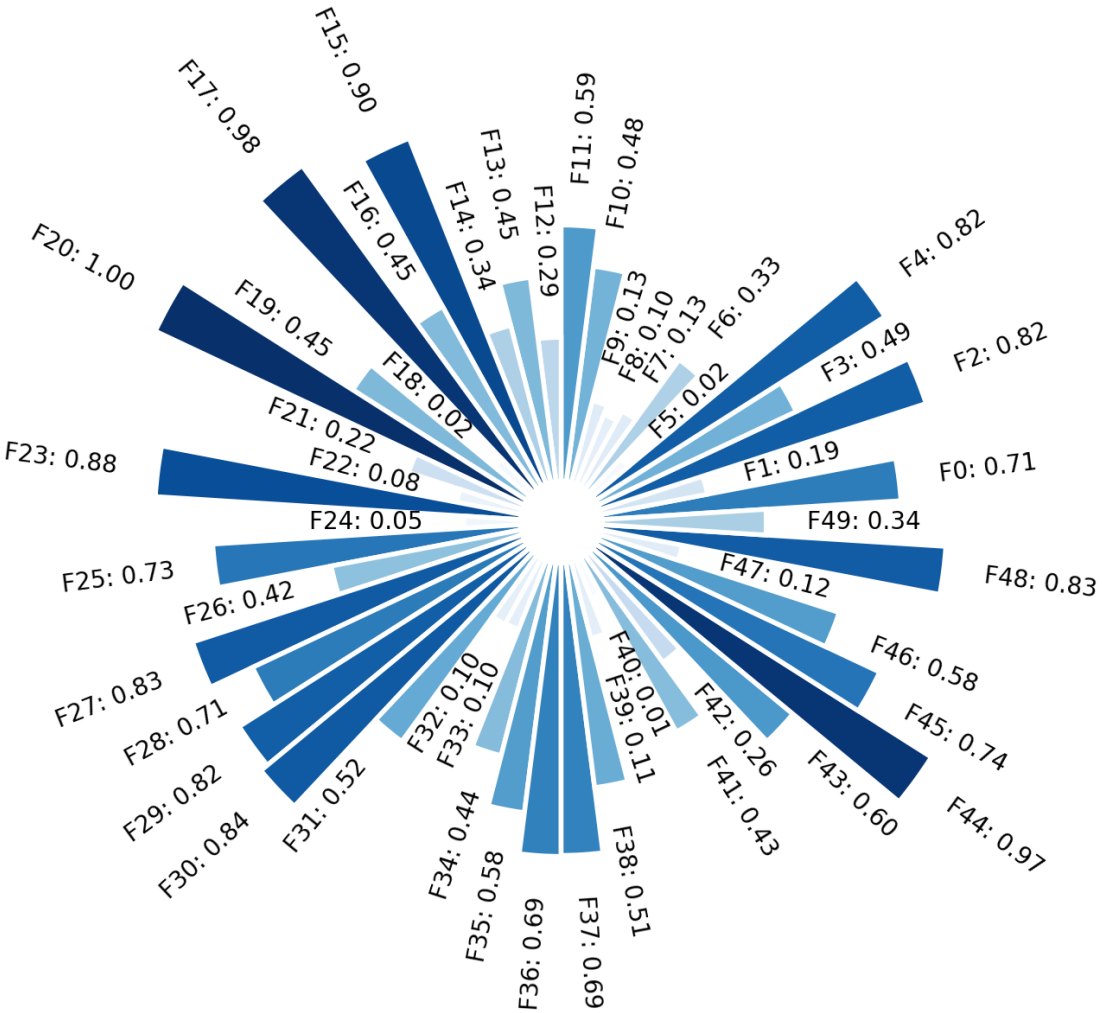
custom color map

```
_ = circular_bar_plot(data, names, color='viridis')
```


F0: 0.71
F1: 0.19
F2: 0.82
F3: 0.49
F4: 0.82
F5: 0.02
F6: 0.33
F7: 0.13
F8: 0.10
F9: 0.13
F10: 0.48
F11: 0.59
F12: 0.29
F13: 0.45
F14: 0.34
F15: 0.90
F16: 0.45
F17: 0.98
F18: 0.02
F19: 0.45
F20: 1.00
F21: 0.22
F22: 0.08
F23: 0.88
F24: 0.05
F25: 0.73
F26: 0.42
F27: 0.83
F28: 0.71
F29: 0.82
F30: 0.84
F31: 0.52
F32: 0.10
F33: 0.10
F34: 0.44
F35: 0.58
F36: 0.69
F37: 0.69
F38: 0.51
F39: 0.11
F40: 0.01
F41: 0.43
F42: 0.26
F43: 0.60
F44: 0.97
F45: 0.74
F46: 0.58
F47: 0.12
F48: 0.83
F49: 0.34

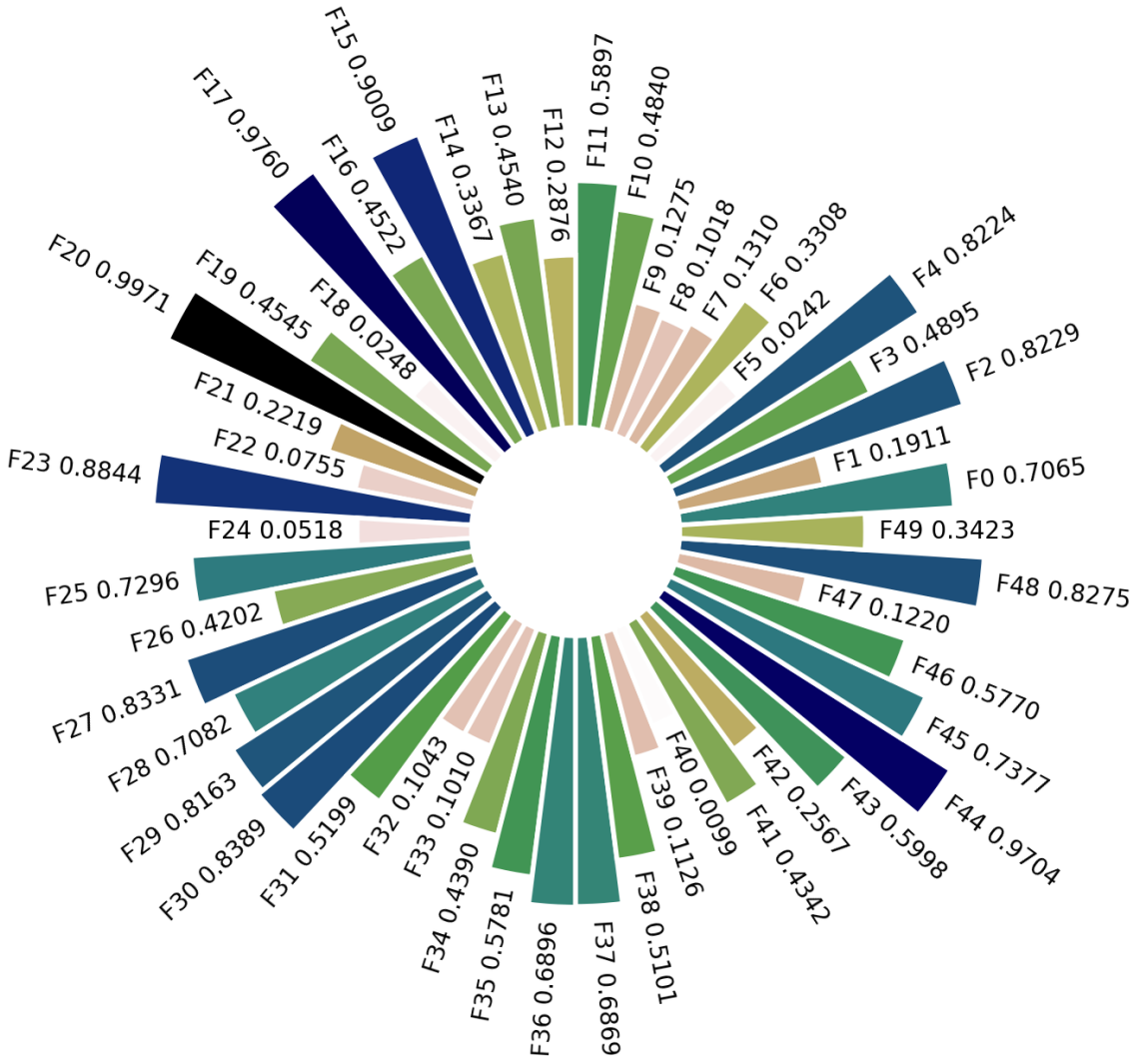
custom min and max range

```
_ = circular_bar_plot(data, names, min_max_range=(1, 10), label_padding=1)
```


F11: 0.59
F10: 0.48
F4: 0.82
F2: 0.82
F0: 0.71
F48: 0.83
F46: 0.58
F45: 0.74
F44: 0.97
F43: 0.60
F42: 0.26
F41: 0.43
F40: 0.01
F39: 0.11
F38: 0.51
F37: 0.69
F36: 0.69
F35: 0.58
F34: 0.44
F33: 0.10
F32: 0.10
F31: 0.52
F30: 0.84
F29: 0.82
F28: 0.71
F27: 0.83
F26: 0.42
F25: 0.73
F24: 0.05
F23: 0.88
F22: 0.08
F21: 0.22
F20: 1.00
F19: 0.45
F18: 0.02
F17: 0.98
F16: 0.45
F15: 0.90
F14: 0.34
F13: 0.45
F12: 0.29
F9: 0.13
F8: 0.10
F7: 0.13
F6: 0.33
F5: 0.02
F3: 0.49
F1: 0.19
F49: 0.34
F47: 0.12

custom label format

```
_ = circular_bar_plot(data, names, label_format='{} {:.4f}')
```


F11 0.5897
F10 0.4840
F9 0.1275
F8 0.1018
F7 0.1310
F6 0.3308
F4 0.8224
F3 0.4895
F2 0.8229
F1 0.1911
F0 0.7065
F49 0.3423
F48 0.8275
F47 0.1220
F46 0.5770
F45 0.7377
F44 0.9704
F43 0.5998
F42 0.2567
F41 0.4342
F40 0.0099
F39 0.1126
F38 0.5101
F37 0.6869
F36 0.6896
F35 0.5781
F34 0.4390
F33 0.1010
F32 0.1043
F31 0.5199
F30 0.8389
F29 0.8163
F28 0.7082
F27 0.8331
F26 0.4202
F25 0.7296
F24 0.0518
F23 0.8844
F22 0.0755
F21 0.2219
F20 0.9971
F19 0.4545
F18 0.0248
F17 0.9760
F16 0.4522
F15 0.9009
F14 0.3367
F13 0.4540
F12 0.2876

Total running time of the script: (0 minutes 5.608 seconds)

6.9 histogram plot

```
from easy_mpl import hist
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from easy_mpl.utils import version_info


version_info()  # print version information of all the packages being used
```

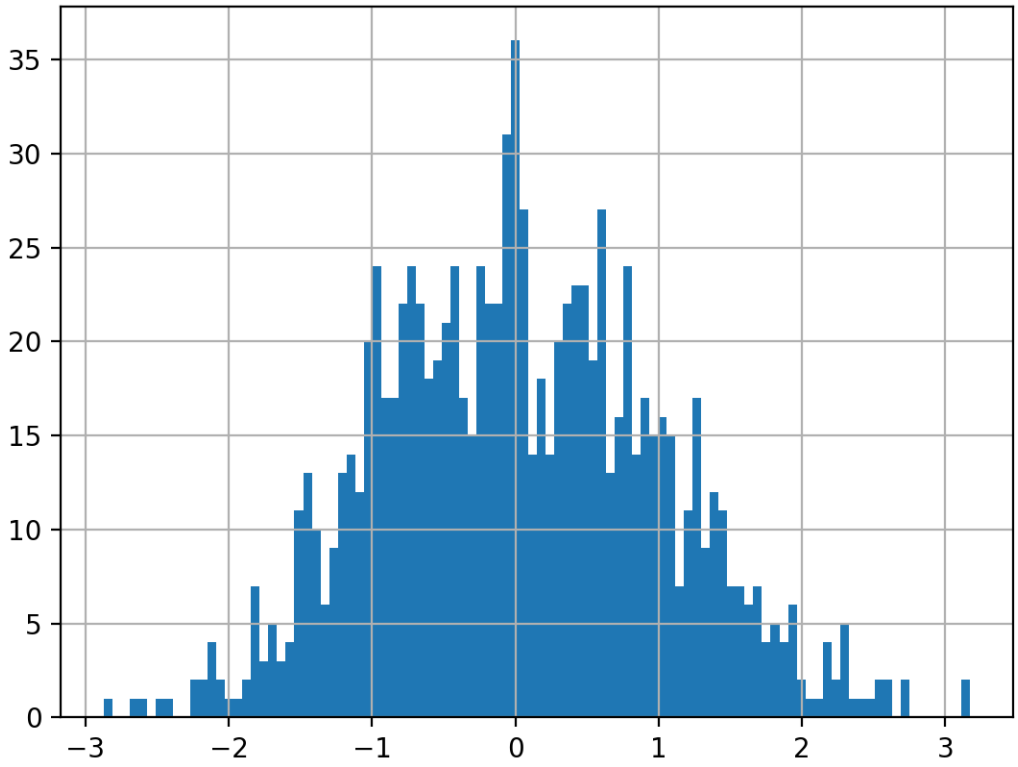
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
f = "https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/datasets/arg_
↪busan.csv"
df = pd.read_csv(f, index_col='index')
cols = ['air_temp_c', 'wat_temp_c', 'sal_psu', 'tide_cm', 'rel_hum', 'pcp12_mm']
```

```
data = np.random.randn(1000)
```

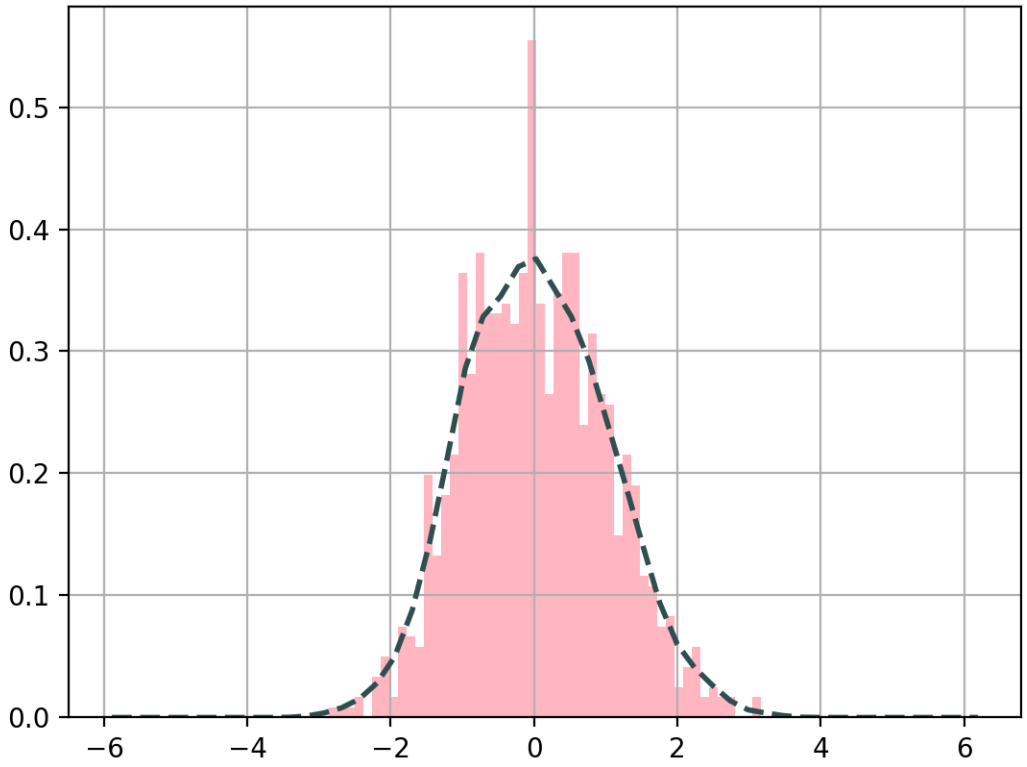
let's start with a basic histogram

```
_ = hist(data, bins = 100)
```



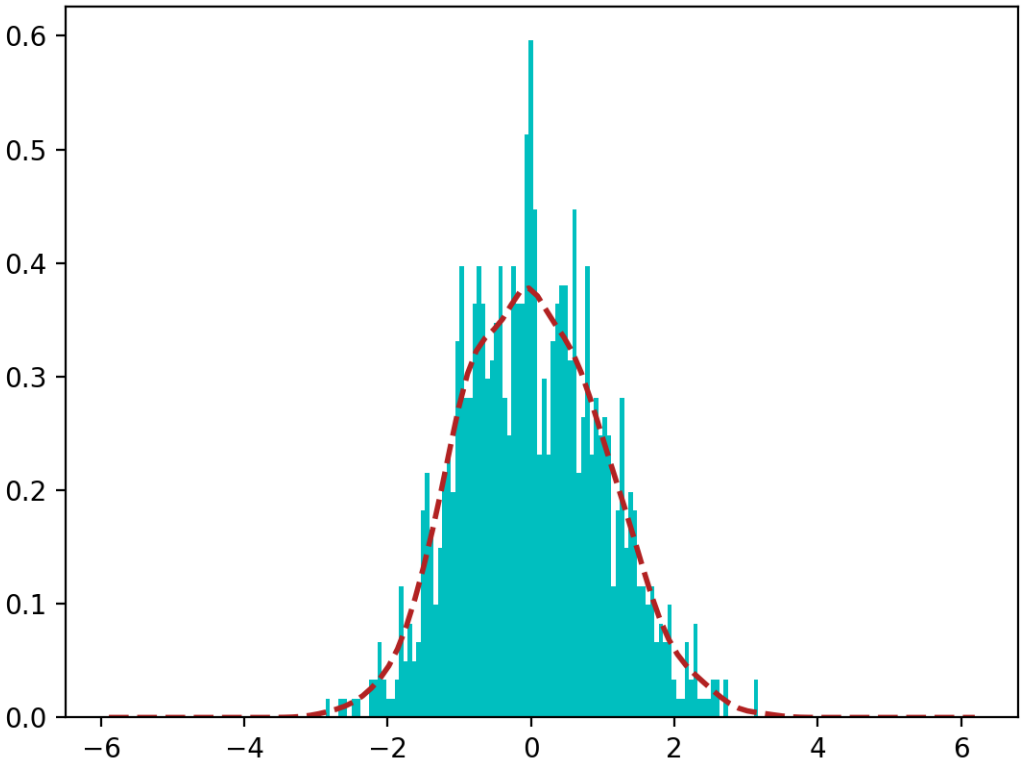
adding KDE and specifying line properties

```
_ = hist(data, add_kde=True, bins=50, color = 'lightpink',
         line_kws={'linestyle': '--',
                   'color': 'darkslategrey',
                   'linewidth': 2.0}
         )
```



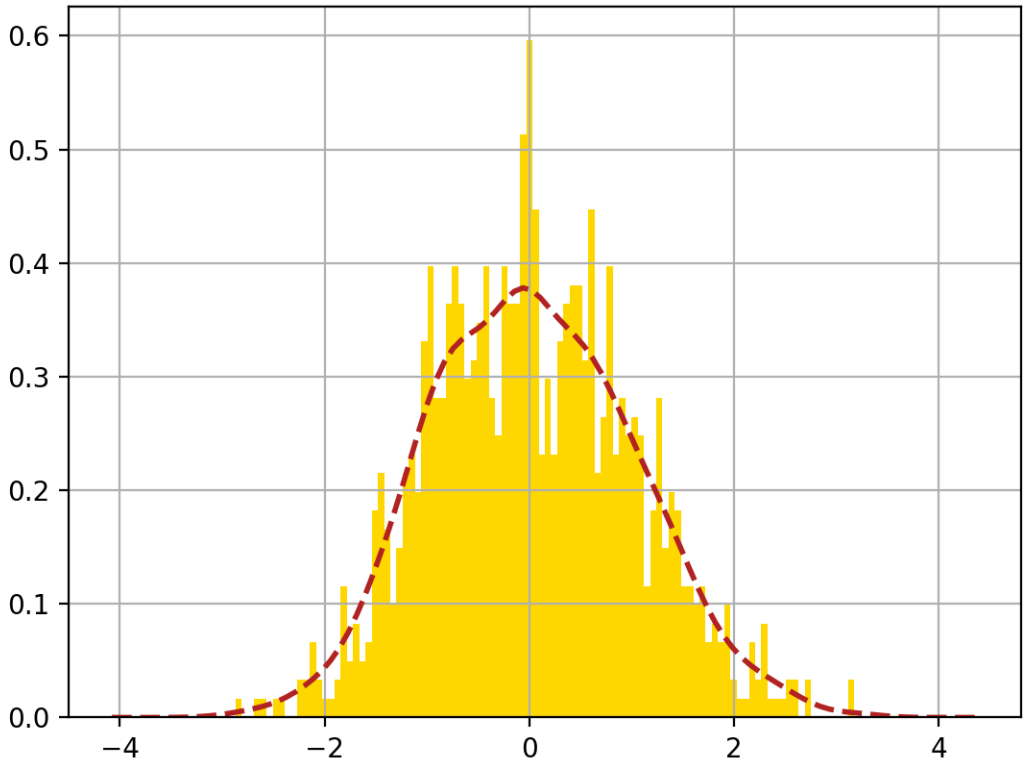
setting grid to False

```
_ = hist(data, bins = 100, grid = False, color = 'c',
         add_kde=True, line_kws={'linestyle': '--',
                                 'color':'firebrick',
                                 'linewidth': 2.0}
         )
```



manipulating kde calculation

```
_ = hist(data, bins = 100, color = 'gold',
         add_kde=True, kde_kws=dict(cut=0.2),
         line_kws={'linestyle': '--',
                   'color':'firebrick',
                   'linewidth': 2.0})
```

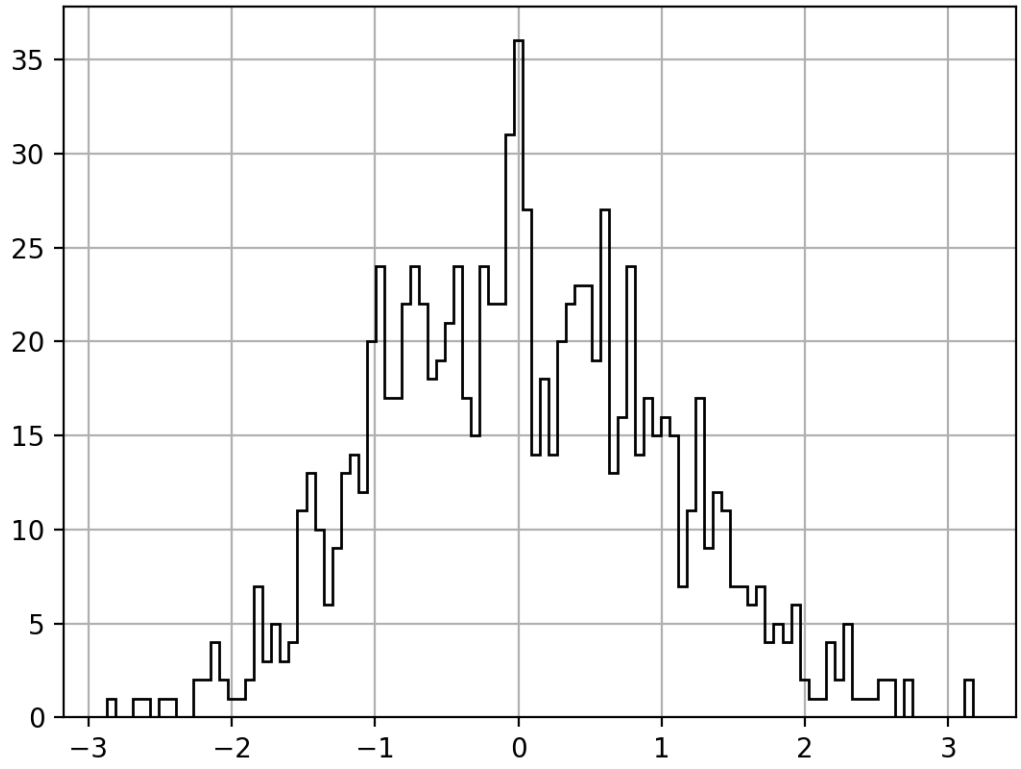


Any argument for matplotlib.hist can be given to hist function for example `color` or `edgecolor`

```
_ = hist(data, bins = 20, linewidth = 0.5,
         edgecolor = "k", grid=False, color='khaki')
```

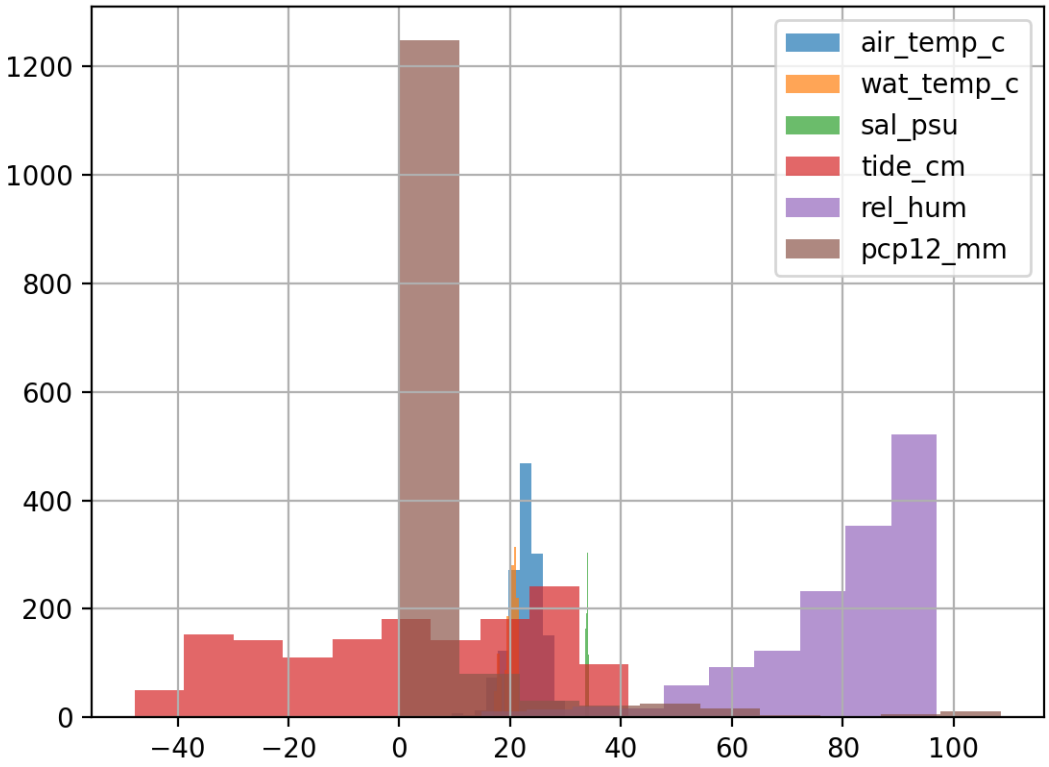
`histtype` defines the type of histogram to show. Here we are using `step` which generates a lineplot that is by default unfilled.

```
_ = hist(data, bins = 100, edgecolor = "k", histtype='step')
```



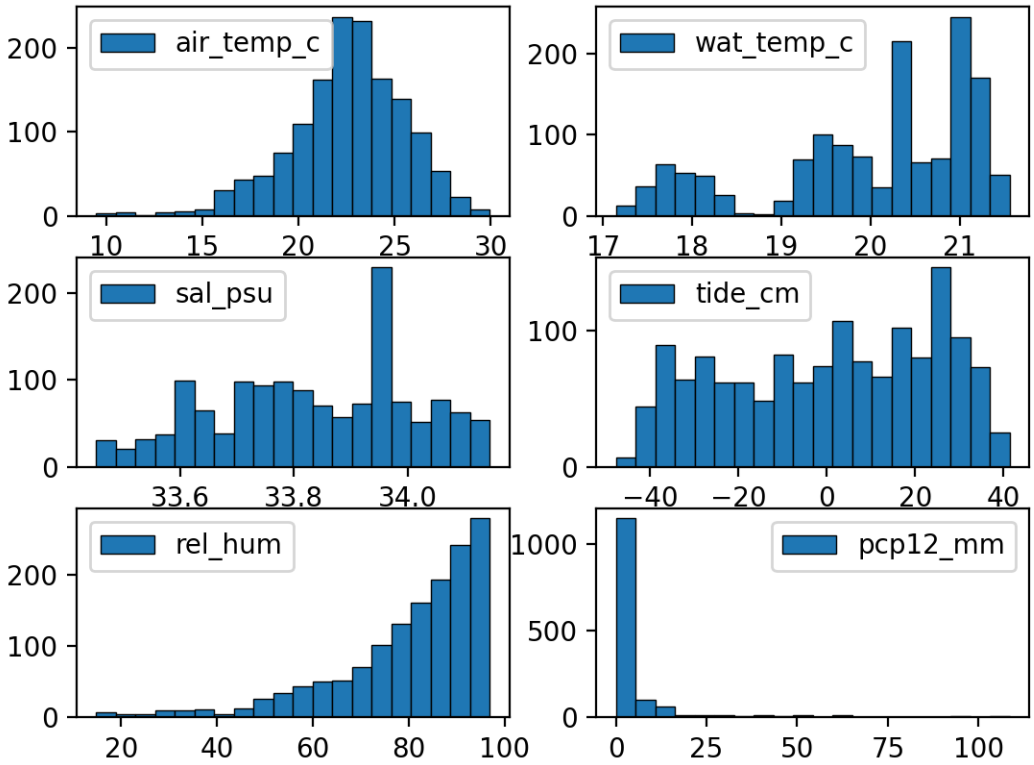
if data contains multiple columns, it will be plotted on same axis

```
_ = hist(df[cols], alpha=0.7)
```



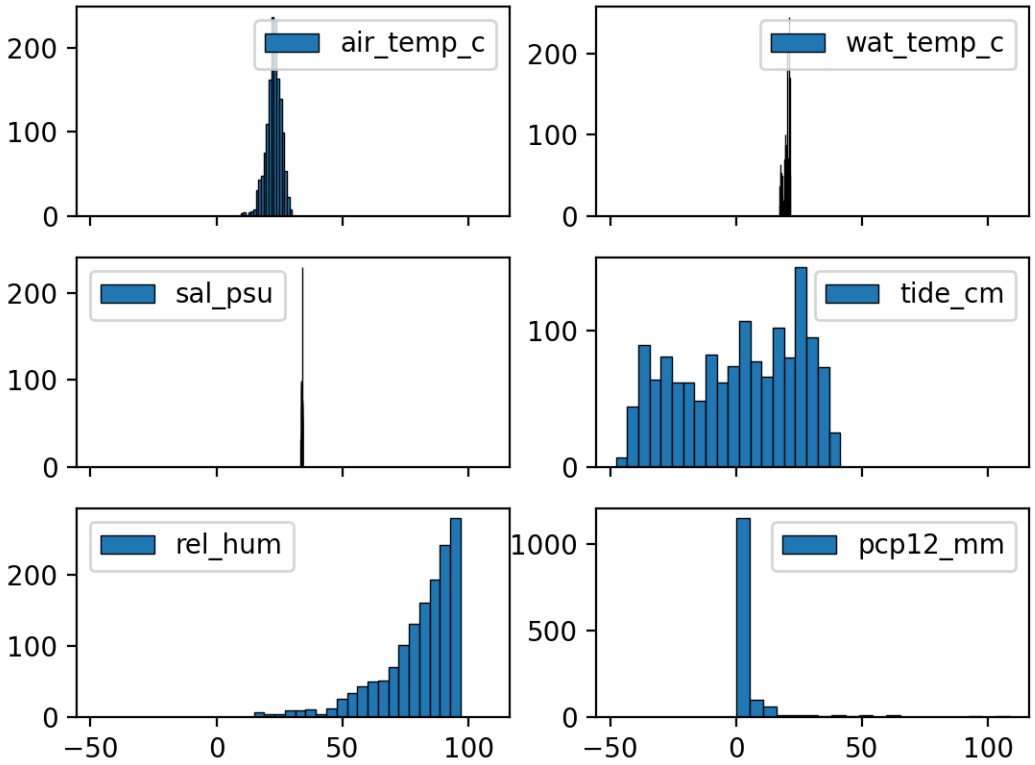
`share_axes` can be set to False to plot multiple columns on different axis.

```
_ = hist(df[cols], share_axes=False,
         bins = 20, linewidth = 0.5, edgecolor = "k", grid=False)
```



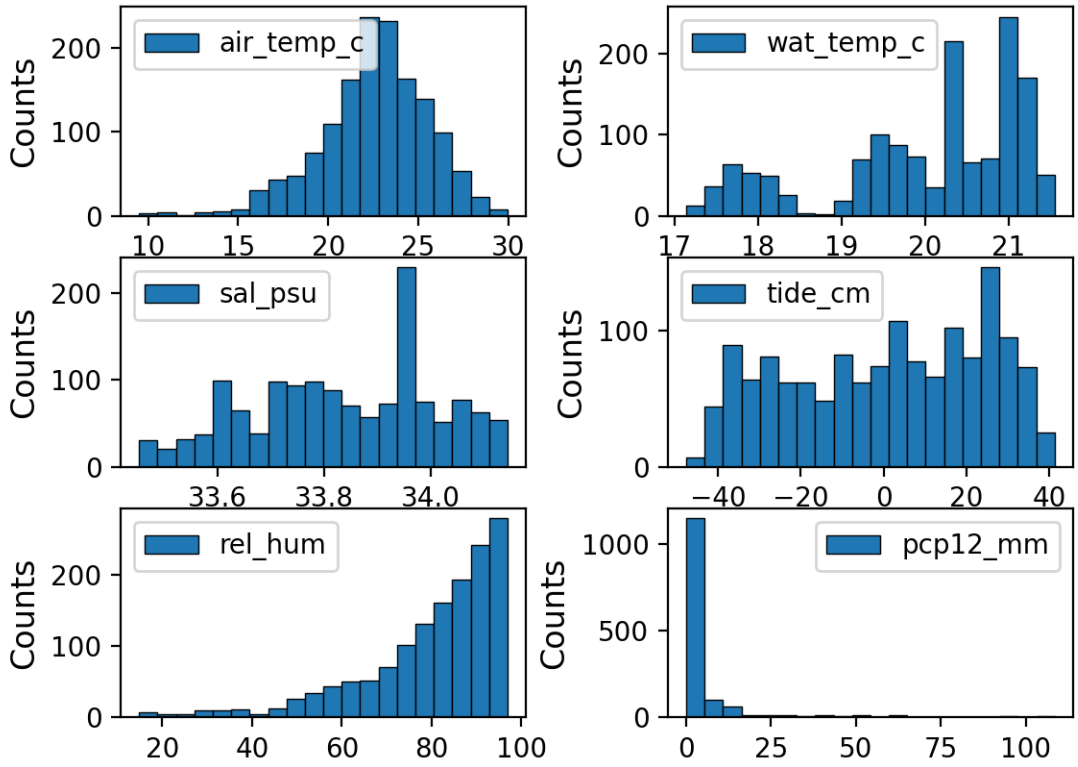
Arguments of subplots can be given to `subplots_kws`

```
_ = hist(df[cols], share_axes=False, subplots_kws={"sharex": "all"},
         bins = 20, linewidth = 0.5, edgecolor = "k", grid=False)
```



`return_axes` can be set to True if we want to further work on current axis

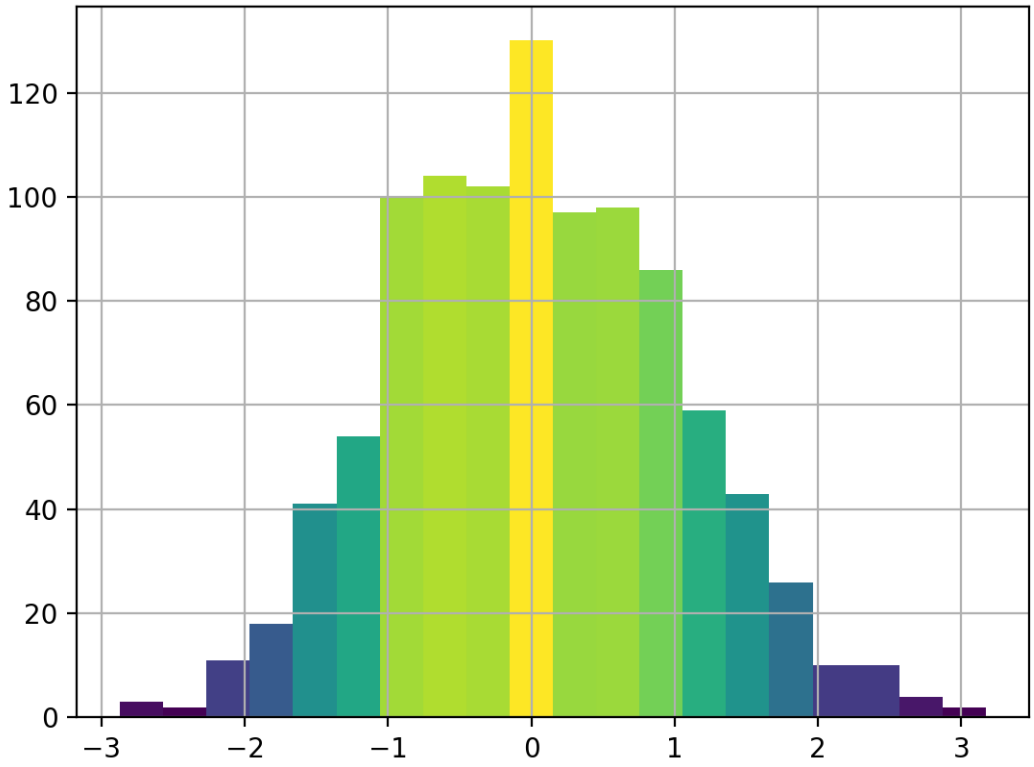
```
outs, axes = hist(df[cols], share_axes=False,
         bins = 20, linewidth = 0.5, edgecolor = "k", grid=False,
                  return_axes=True, show=False)
print(f"{len(outs)} {len(axes)}")
for idx, ax in enumerate(axes):
    ax.set_ylabel('Counts', fontsize=12)
plt.subplots_adjust(wspace=0.35)
plt.show()
```




```
6 6
```

we can further modify colors for better understanding and correct readings for example in this plot, the color represent the frequency of data.

```
n, bins, patches = hist(data, bins = 20, show=False)

# Setting color
f = ((n ** (1 / 2)) / n.max())
norm = colors.Normalize(f.min(), f.max())

for thisfrac, thispatch in zip(f, patches):
    color = plt.cm.viridis(norm(thisfrac))
    thispatch.set_facecolor(color)

plt.show()
```



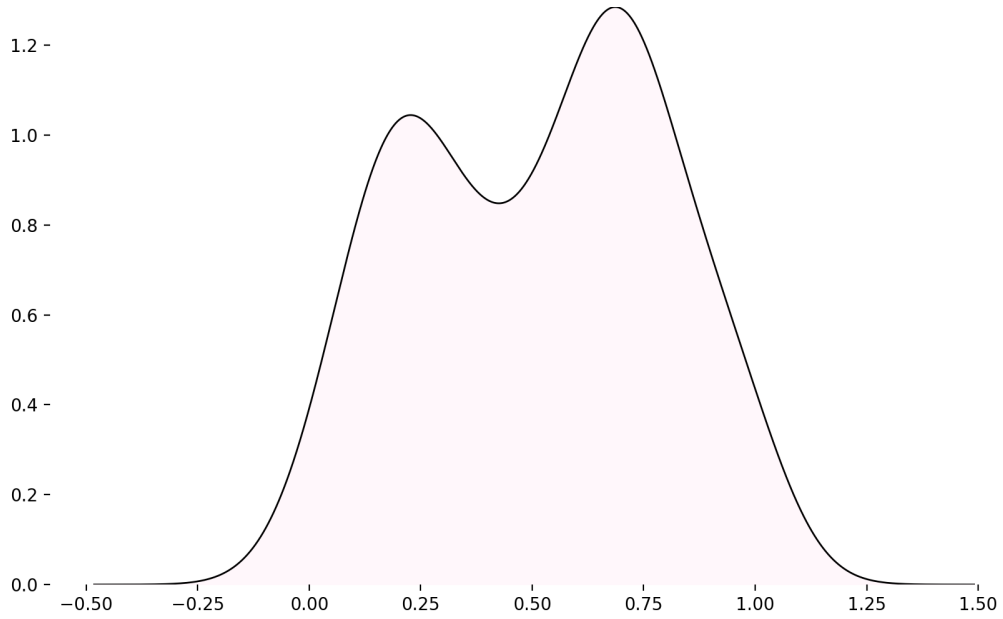
Total running time of the script: (0 minutes 4.477 seconds)

6.10 ridge plot

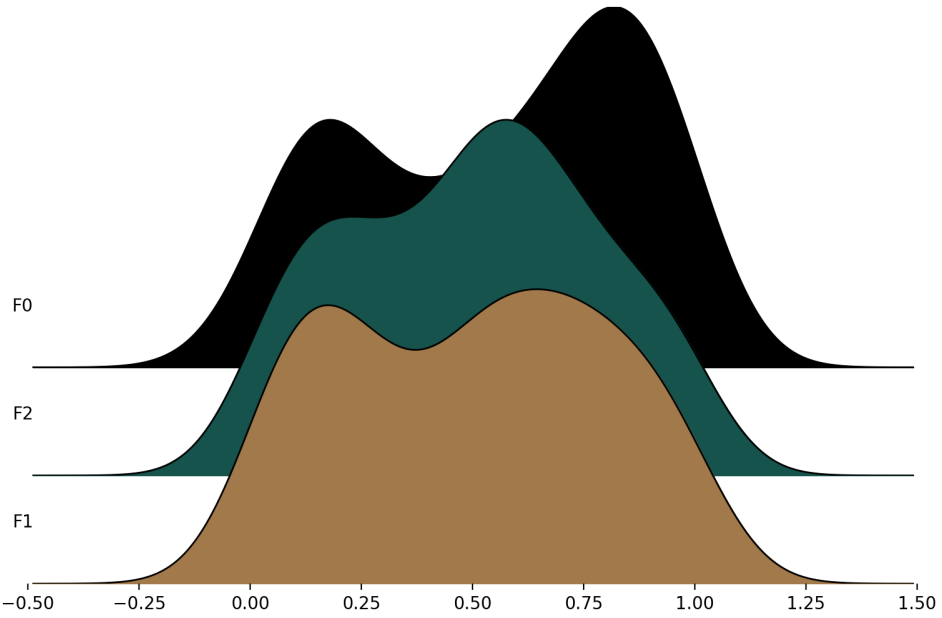
```
# sphinx_gallery_thumbnail_number = 2

import numpy as np
import pandas as pd
from easy_mpl import ridge
import matplotlib.pyplot as plt
from easy_mpl.utils import version_info

version_info()  # print version information of all the packages being used
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
data_ = np.random.random(size=100)
_ = ridge(data_)
```

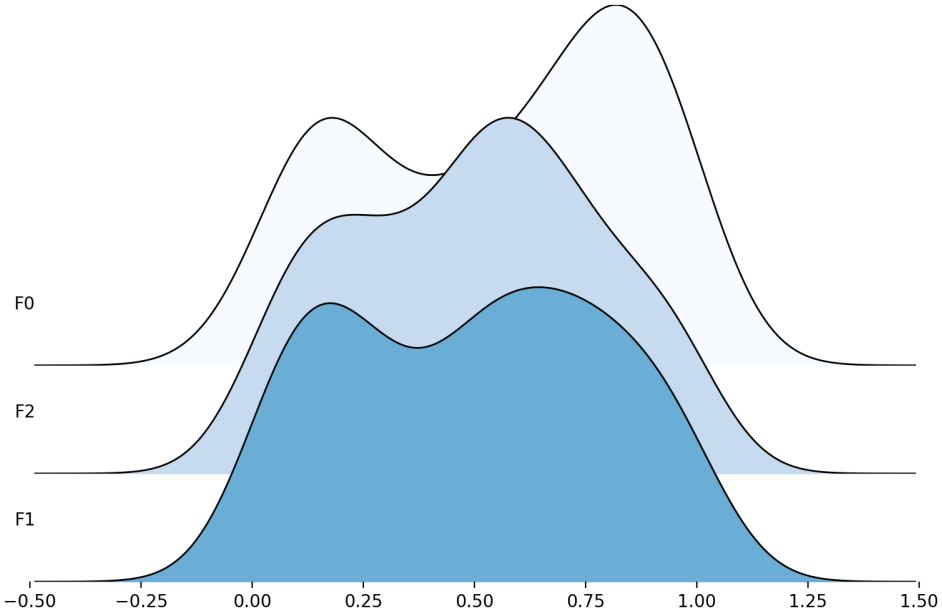



```
data_ = np.random.random((100, 3))
_ = ridge(data_)
```



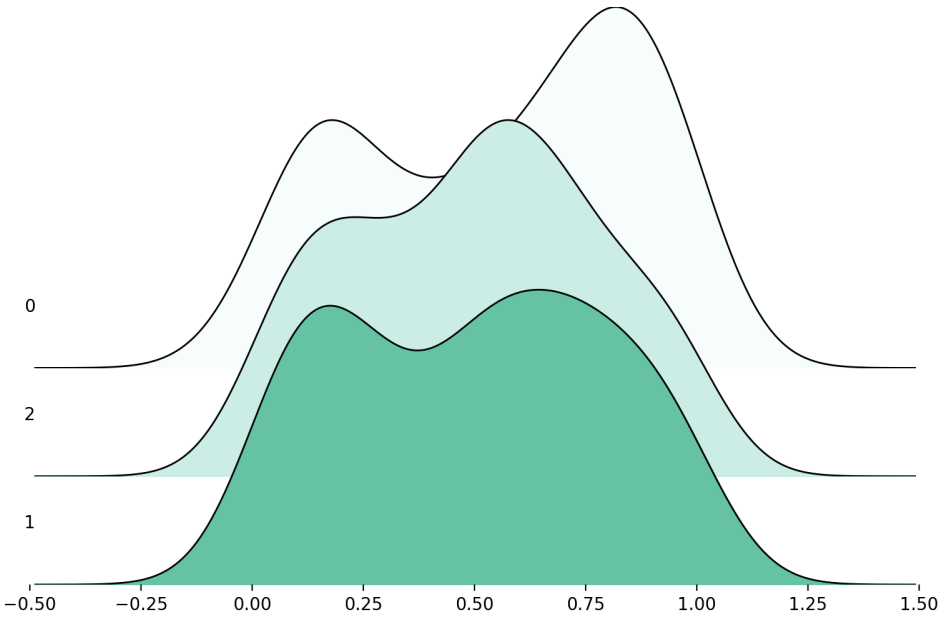
specifying colormap

```
_ = ridge(data_, color="Blues")
```



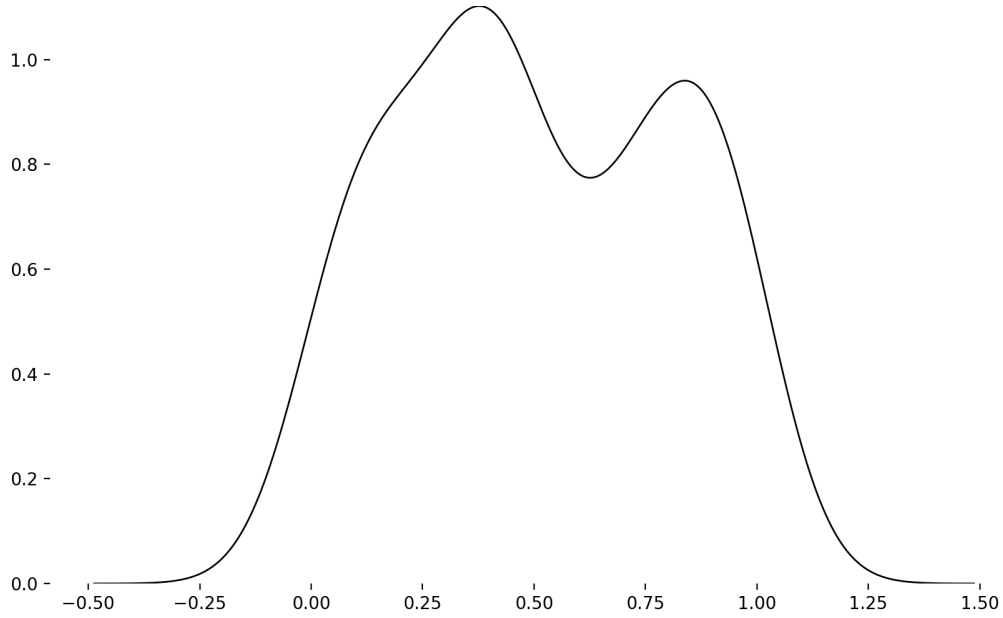
The data can also be in the form of pandas DataFrame

```
_ = ridge(pd.DataFrame(data_))
```



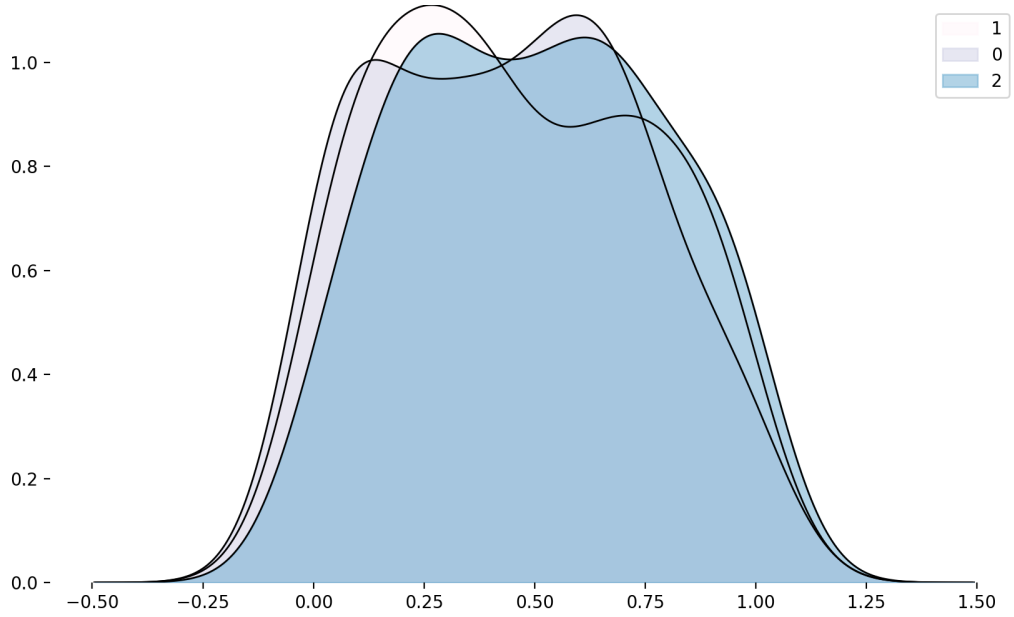
if we don't want to fill the ridge, we can specify the color as white

```
_ = ridge(np.random.random(100), color=["white"])
```



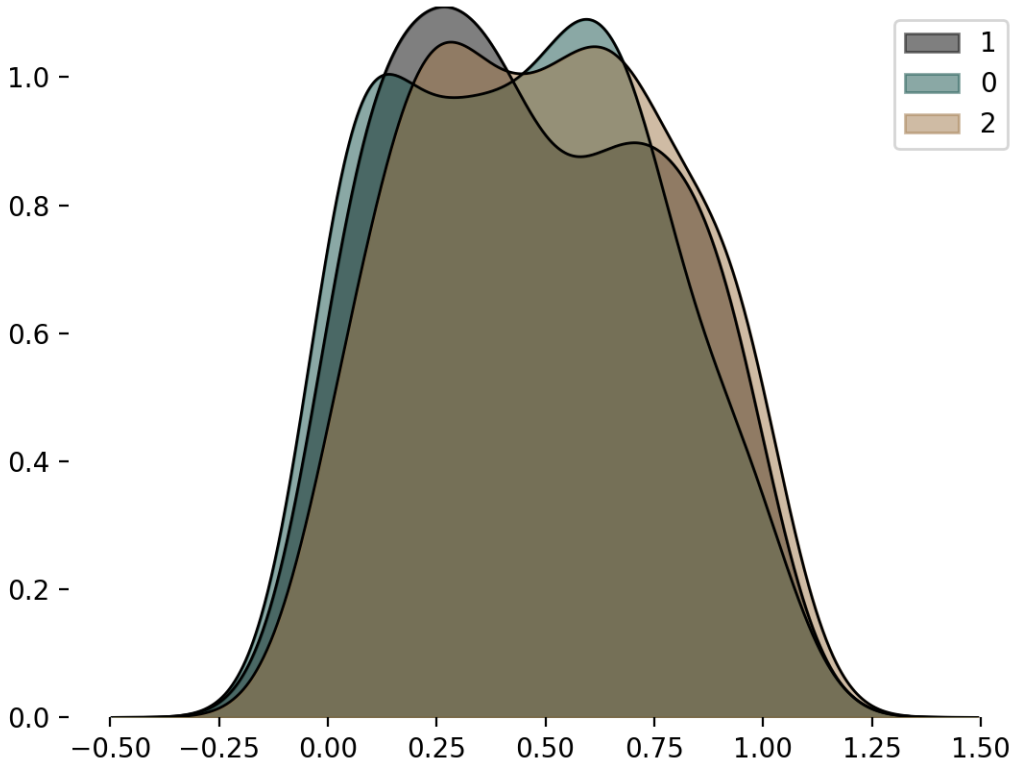
we can draw all the ridges on same axes as below

```
df = pd.DataFrame(np.random.random((100, 3)), dtype='object')
_ = ridge(df, share_axes=True, fill_kws={"alpha": 0.5})
```



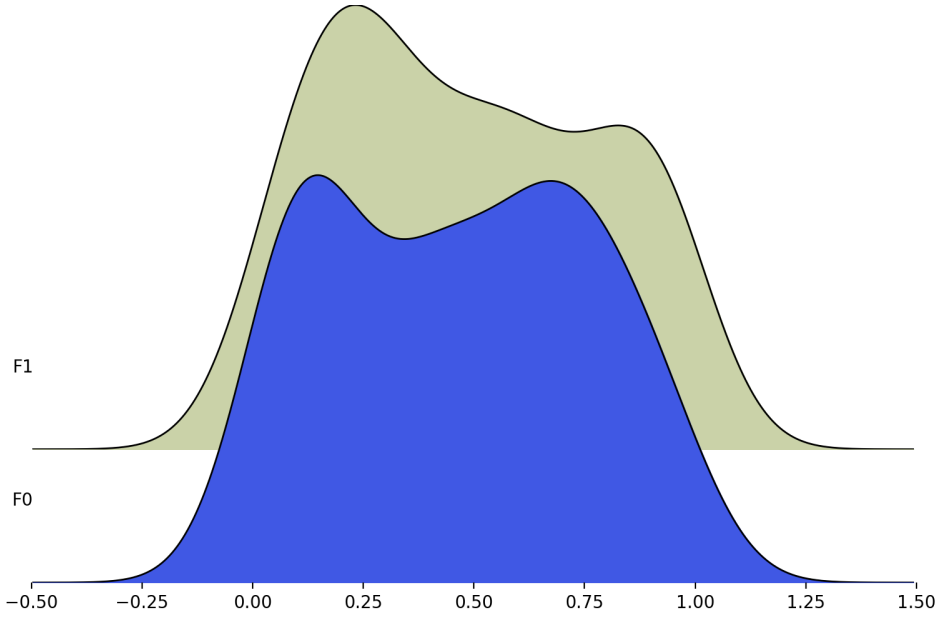
we can also provide an existing axes to plot on

```
_, ax = plt.subplots()
_ = ridge(df, ax=ax, fill_kws={"alpha": 0.5})
```



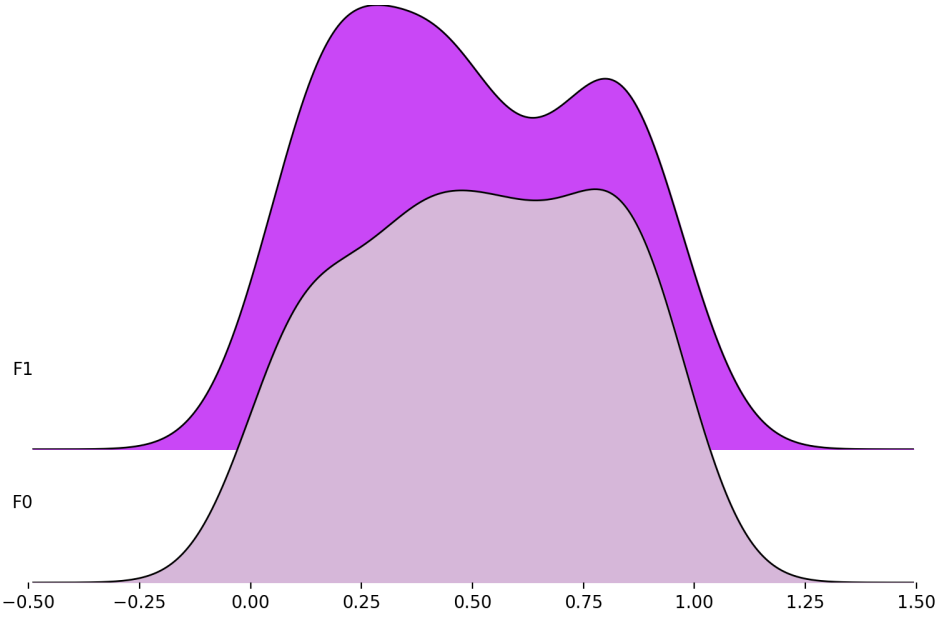
The data can also be in the form of list of arrays

```
x1 = np.random.random(100)
x2 = np.random.random(100)
_ = ridge([x1, x2], color=np.random.random((3, 2)))
```



The length of arrays need not to be constant/same. We can use arrays of different lengths

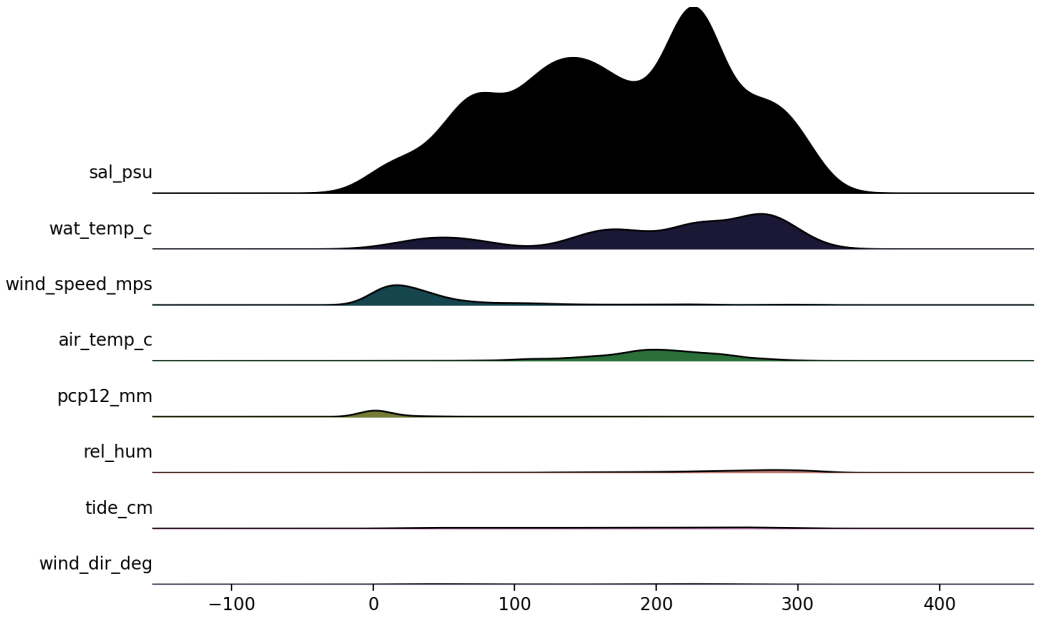
```
x1 = np.random.random(100)
x2 = np.random.random(90)
_ = ridge([x1, x2], color=np.random.random((3, 2)))
```



```
f = "https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/datasets/arg_
↪busan.csv"
df = pd.read_csv(f, index_col='index')
print(df.shape)
df.head()
```

```
(1446, 25)
```

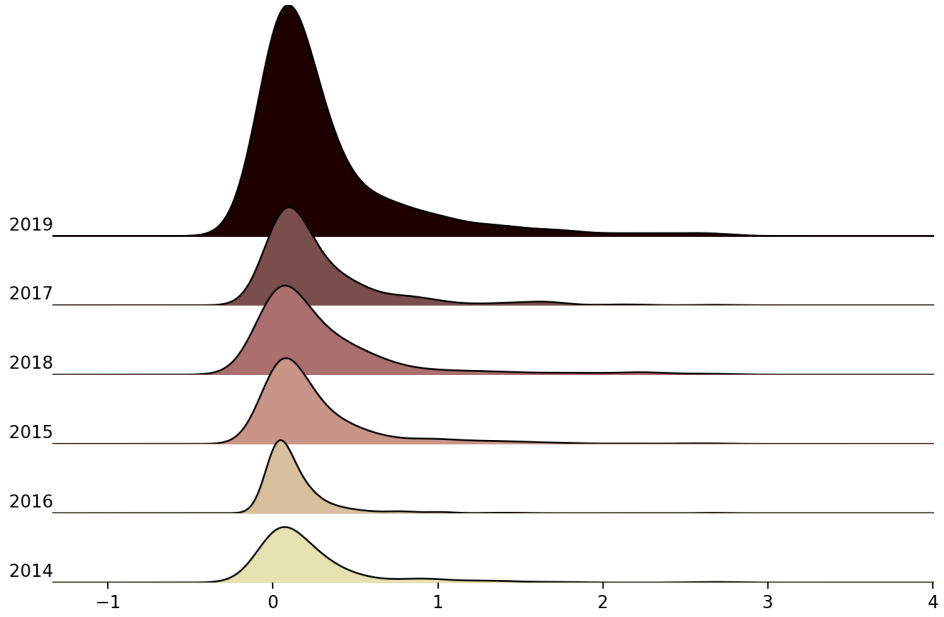
```
cols = ['air_temp_c',
        'wat_temp_c',
        'sal_psu',
        'tide_cm',
        'rel_hum',
        'pcp12_mm',
        'wind_dir_deg',
        'wind_speed_mps'
        ]

_ = ridge(df[cols])
```




```
f = 'https://media.githubusercontent.com/media/HakaiInstitute/essd2021-hydromet-
↪datapackage/main/2013-2019_Discharge1015_5min.csv'
df = pd.read_csv(f)
df.index = pd.to_datetime(df.pop('Datetime'))
print(df.shape)
df.head()
```

```
(543568, 12)
```

```
groupby_year = df.groupby(lambda x: x.year)

_ = ridge(
    [grp['Qrate'].resample('D').interpolate(method='linear') for _, grp in groupby_year],
    labels=[name for name, _ in groupby_year],
    )
```



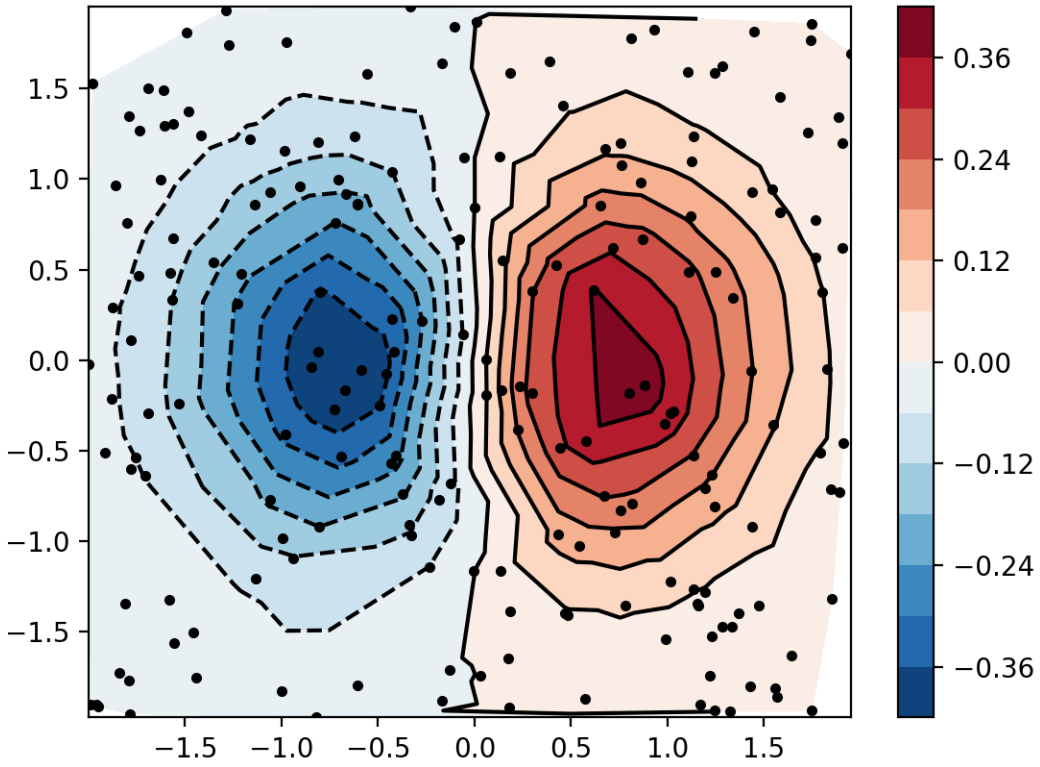
Total running time of the script: (0 minutes 7.051 seconds)

6.11 contour plot

```
from easy_mpl import contour
import numpy as np
from easy_mpl.utils import version_info

version_info()
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

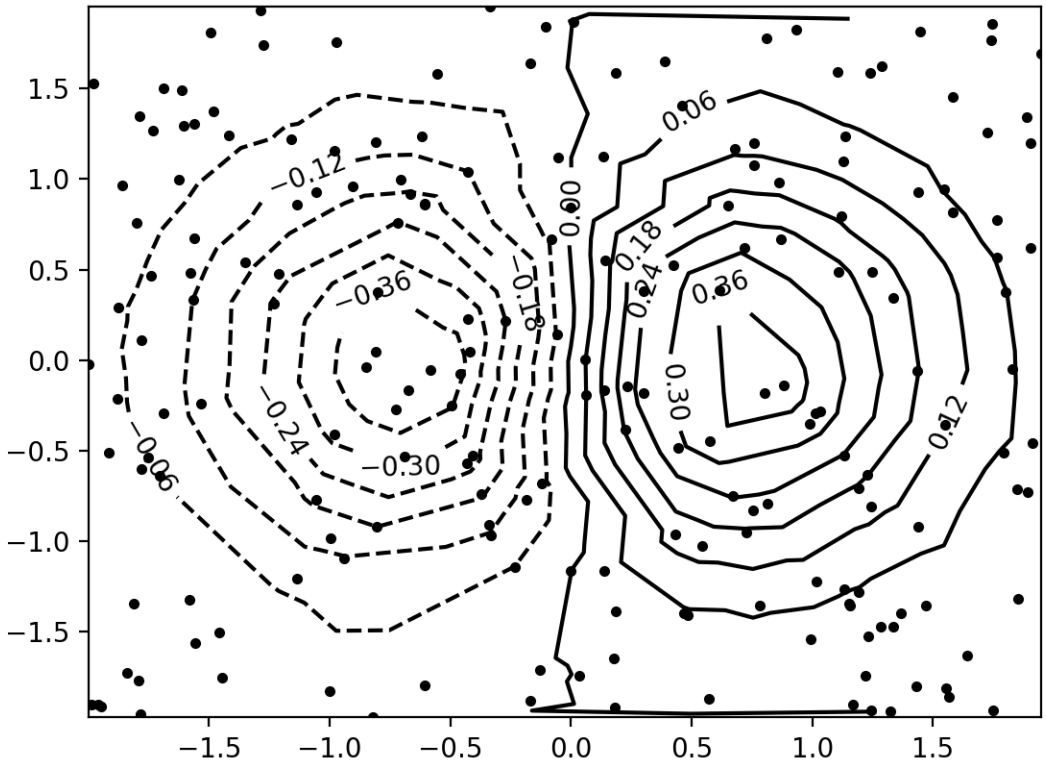
```
_x = np.random.uniform(-2, 2, 200)
_y = np.random.uniform(-2, 2, 200)
_z = _x * np.exp(-_x**2 - _y**2)
contour(_x, _y, _z, fill_between=True, show_points=True)
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/envs/latest/lib/python3.9/
↪site-packages/easy_mpl/_contour.py:115: UserWarning: The following kwargs were not␣
↪used by contour: 'linewidth'
  CS = ax.tricontour(x, y, z, **_kws)

<Axes: >
```

show contour labels

```
contour(_x, _y, _z, label_contours=True, show_points=True)
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/envs/latest/lib/python3.9/
↪site-packages/easy_mpl/_contour.py:115: UserWarning: The following kwargs were not␣
↪used by contour: 'linewidth'
  CS = ax.tricontour(x, y, z, **_kws)

<Axes: >
```

Total running time of the script: (0 minutes 0.666 seconds)

6.12 pie plot

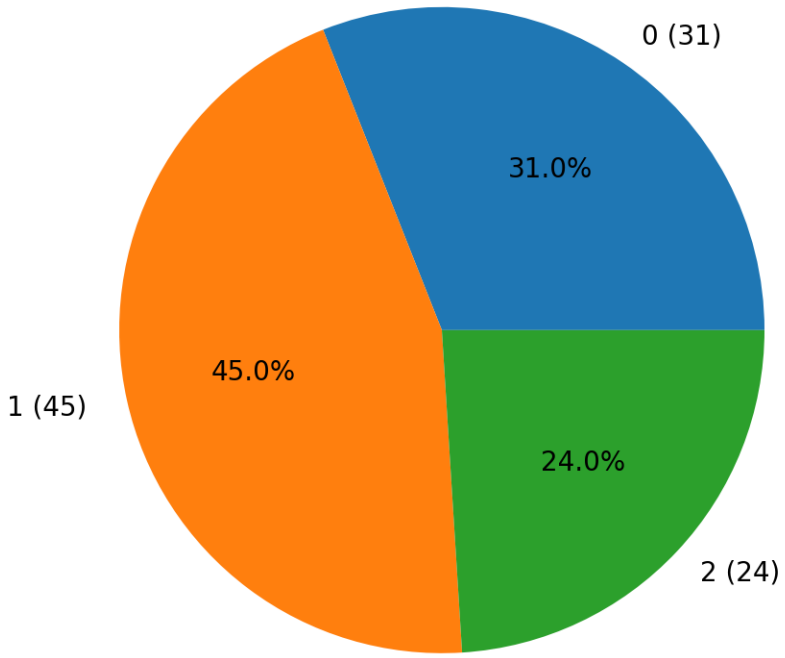
```
import numpy as np
from easy_mpl import pie
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from easy_mpl.utils import version_info, map_array_to_cmap

version_info()

# sphinx_gallery_thumbnail_number = 5
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

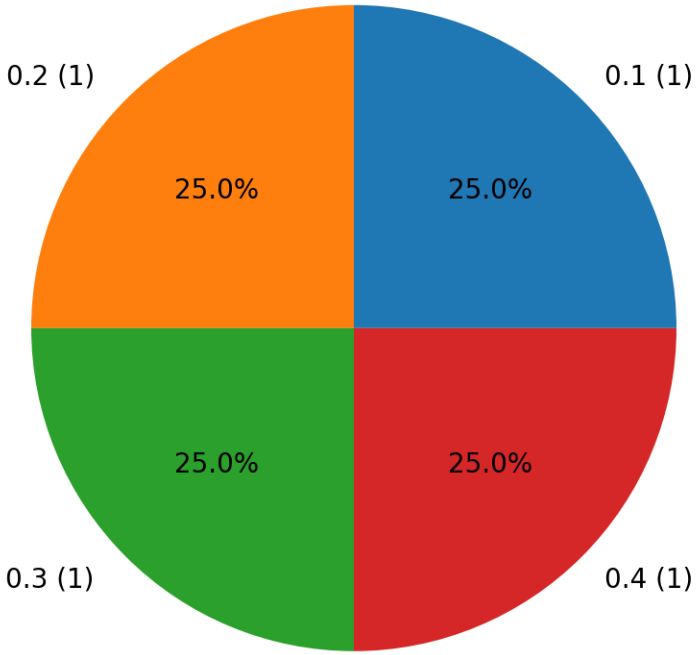
(continues on next page)

(continued from previous page)

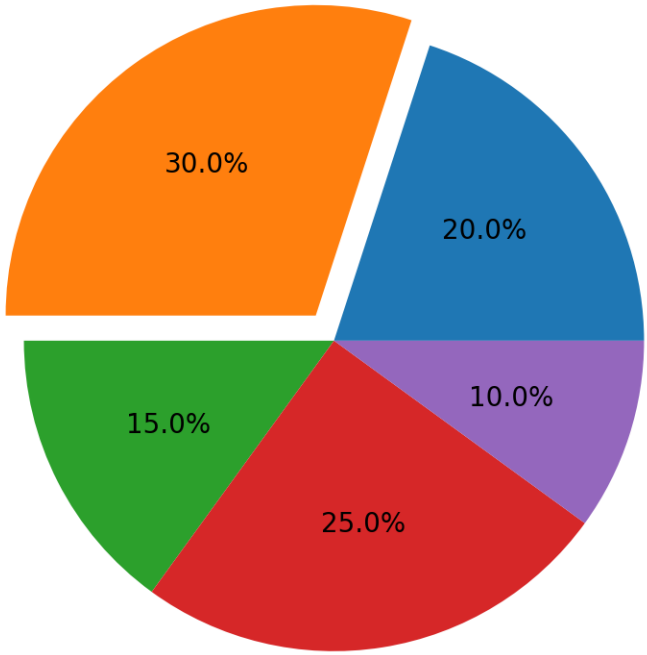
```
_ = pie(np.random.randint(0, 3, 100))
```




```
_ = pie([0.2, 0.3, 0.1, 0.4])
```

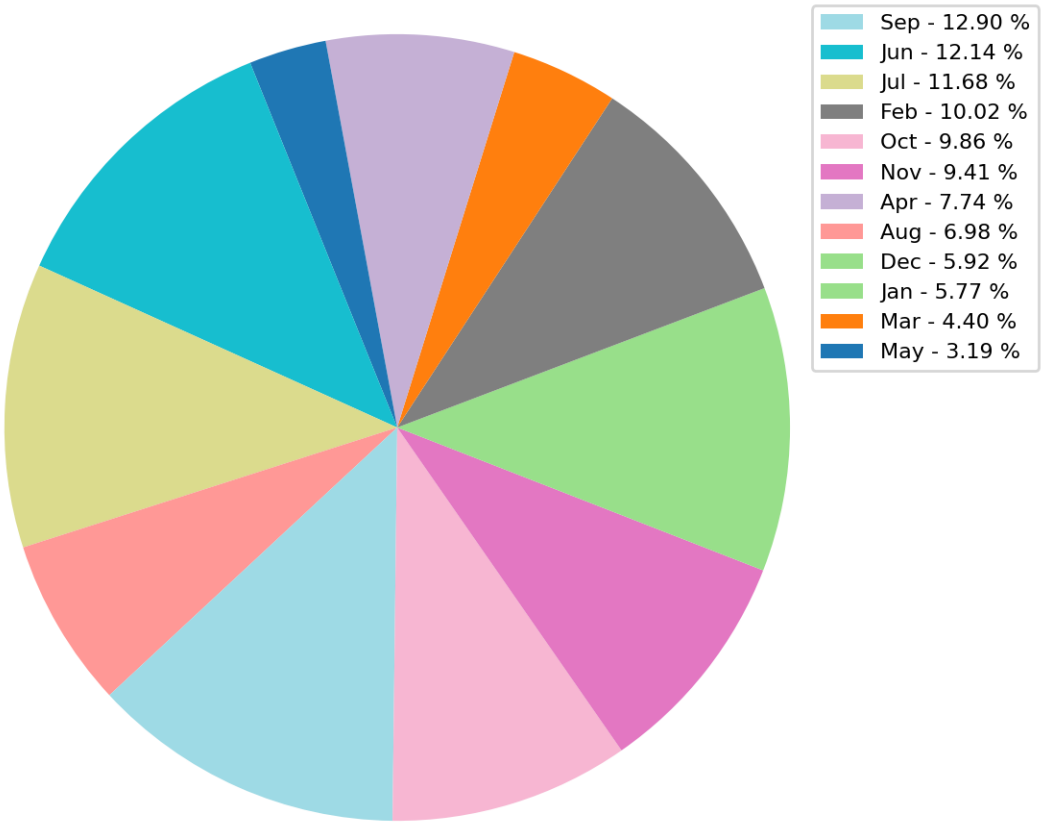



```
# to explode 0.3
explode = (0, 0.1, 0, 0, 0)
_ = pie(fractions=[0.2, 0.3, 0.15, 0.25, 0.1], explode=explode)
```

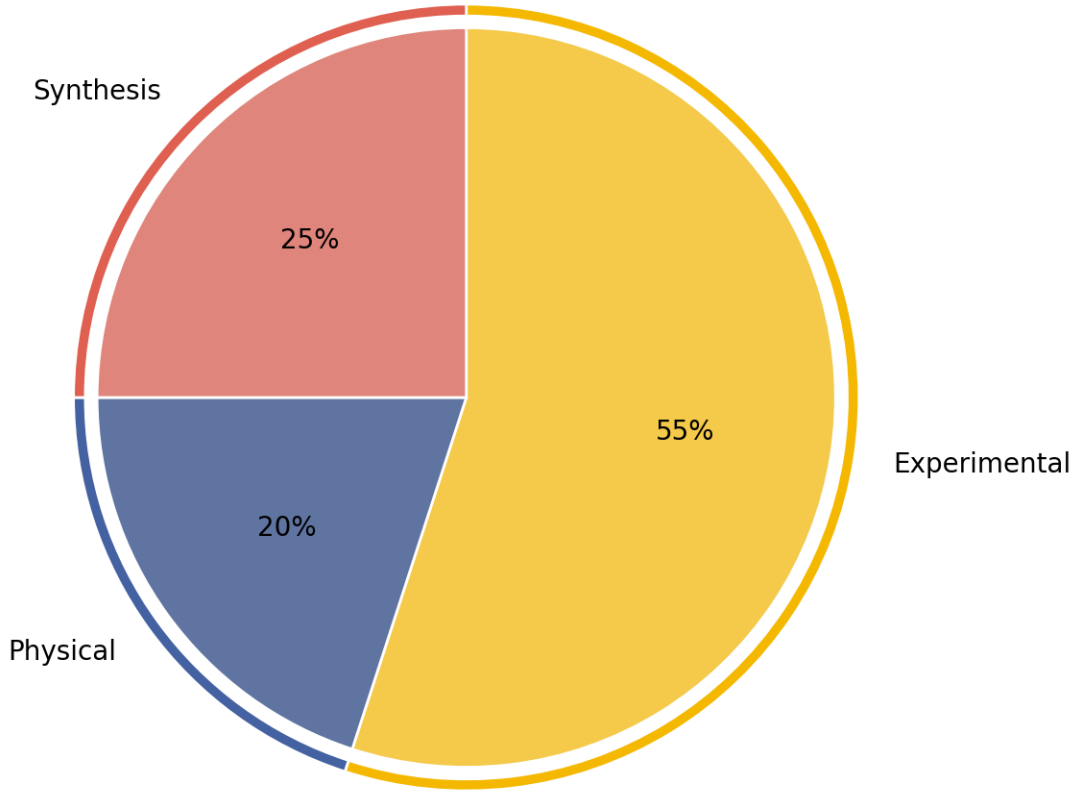


Specifying colors for each section of pie chart

```
rng = np.random.default_rng(313)
labels = ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct', 'Nov','Dec']
y = rng.integers(10, 100, 12)
colors, _ = map_array_to_cmap(y, cmap="tab20")
percent = 100.*y/y.sum()

outs = pie(fractions=percent, autopct=None,
           colors=colors, show=False)
patches, texts = outs
labels = ['{0} - {1:1.2f} %'.format(i,j) for i,j in zip(labels, percent)]

patches, labels, dummy =  zip(*sorted(zip(patches, labels, y),
                                      key=lambda x: x[2],
                                      reverse=True))

plt.legend(patches, labels, bbox_to_anchor=(1.1, 1.),
           fontsize=8)
plt.tight_layout()
plt.show()
```




```
seg_colors = ["#F5B800", "#4461A1", "#DF5F50"]

# Change the saturation of seg_colors to 70% for the interior segments
rgb = mcolors.to_rgba_array(seg_colors)[:,:-1]
hsv = mcolors.rgb_to_hsv(rgb)
hsv[:,1] = 0.7 * hsv[:, 1]
interior_colors = mcolors.hsv_to_rgb(hsv)

pie(fractions=[0.55, 0.2, 0.25],  colors=seg_colors,
       labels=['Experimental', 'Physical', 'Synthesis'],
       wedgeprops=dict(edgecolor="w", width=0.03), radius=1,
autopct=None,
       startangle=90, counterclock=False, show=False)

pie(fractions=[0.55, 0.2, 0.25], colors=interior_colors,
       autopct='%1.0f%%',
       wedgeprops=dict(edgecolor="w"), radius=1-2*0.03,
       startangle=90, counterclock=False, ax=plt.gca(), show=False)

plt.tight_layout()
plt.show()
```



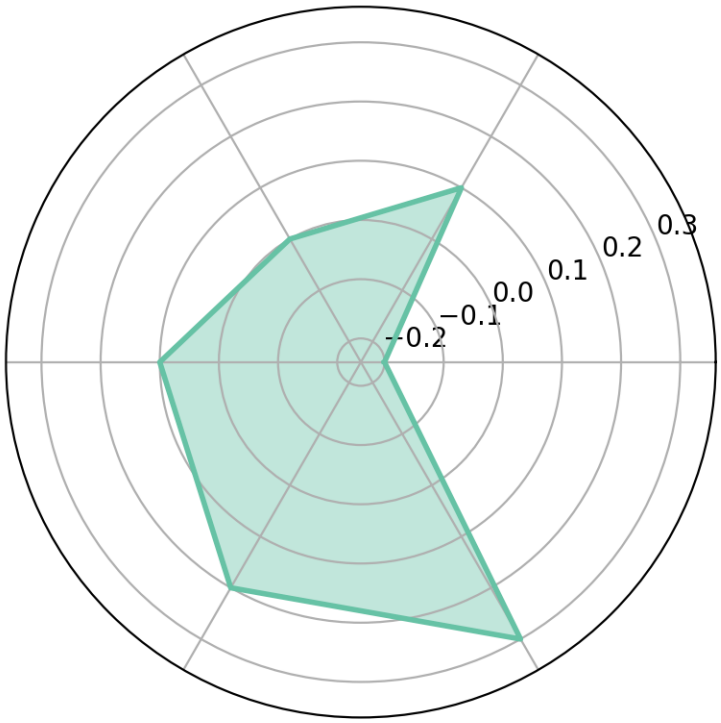
Total running time of the script: (0 minutes 0.871 seconds)

6.13 spider plot

```
# sphinx_gallery_thumbnail_number = -2

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from easy_mpl import spider_plot
from easy_mpl.utils import version_info, create_subplots

version_info()
```

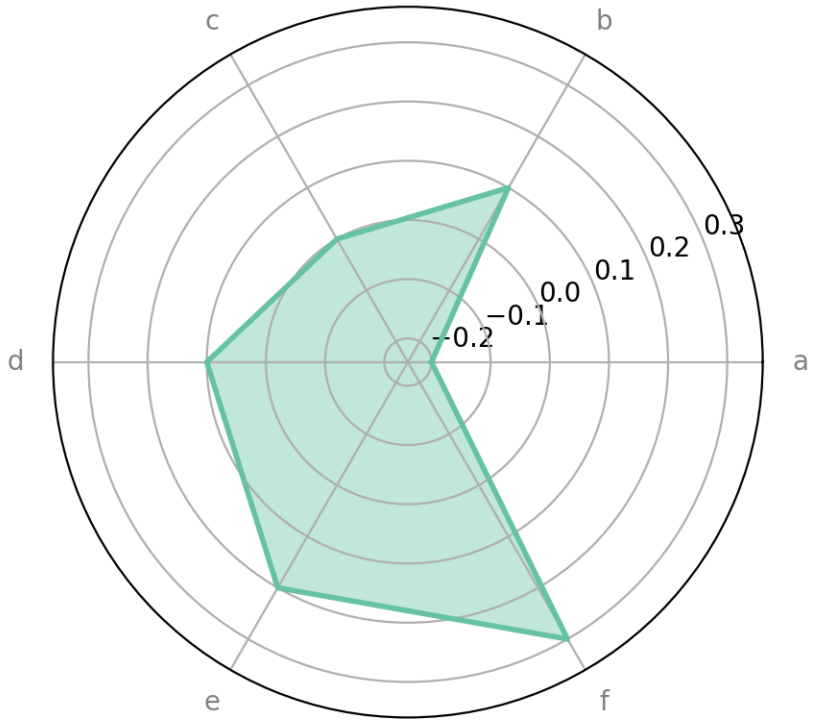
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
values = [-0.2, 0.1, 0.0, 0.1, 0.2, 0.3]
_ = spider_plot(data=values)
```



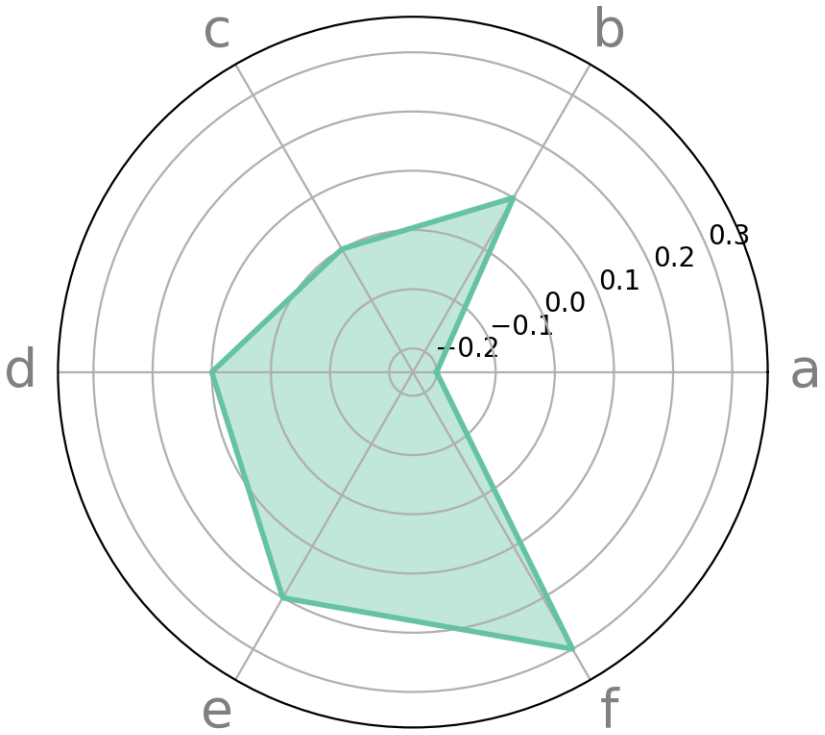
specifying labels

```
labels = ['a', 'b', 'c', 'd', 'e', 'f']
_ = spider_plot(data=values, tick_labels=labels)
```



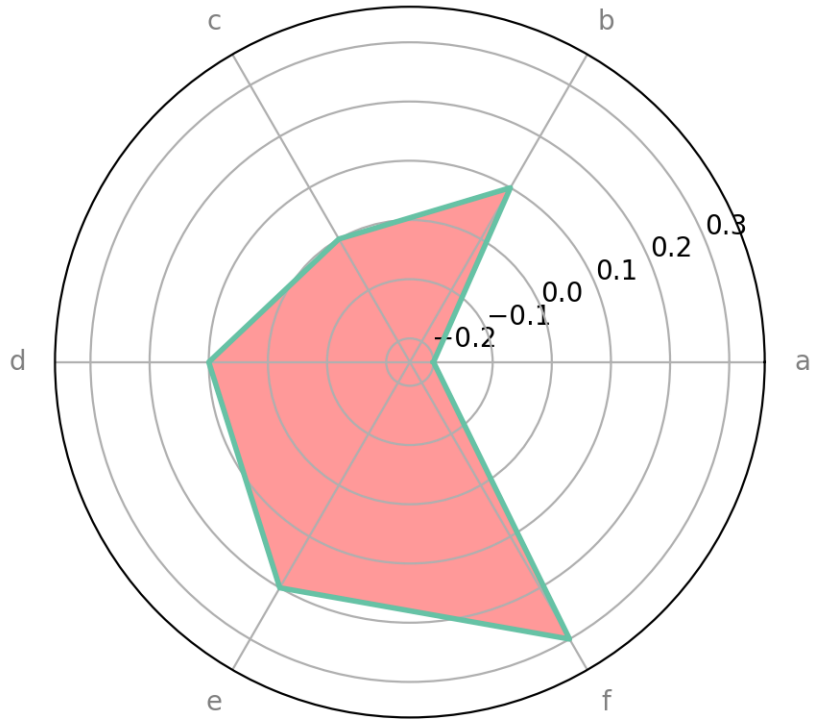
specifying tick size

```
_ = spider_plot(values, labels, xtick_kws={'size': 20})
```



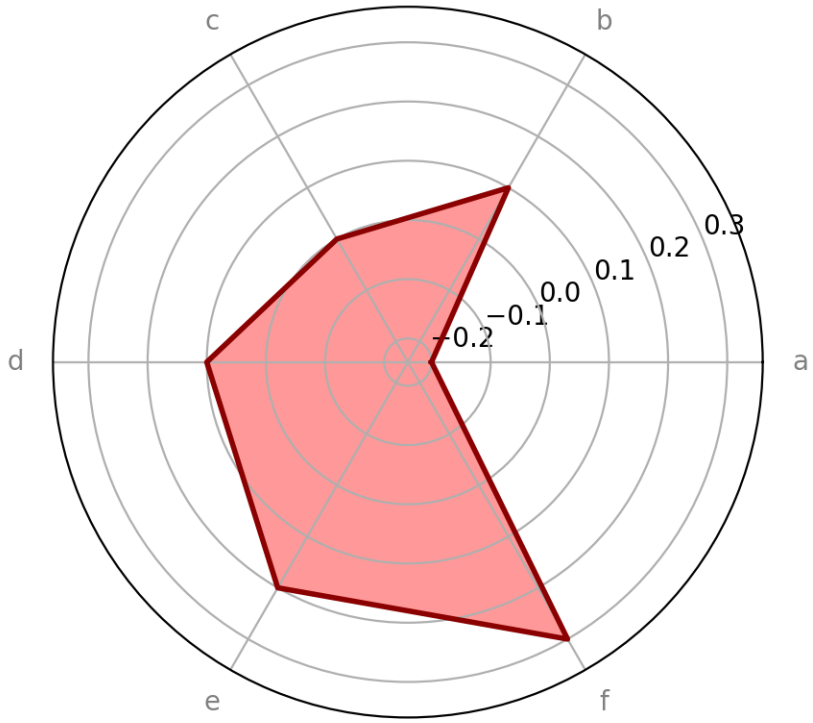
specifying colors on our own

```
_ = spider_plot(values, labels, fill_kws={'color':'r'})
```



specifying outline color

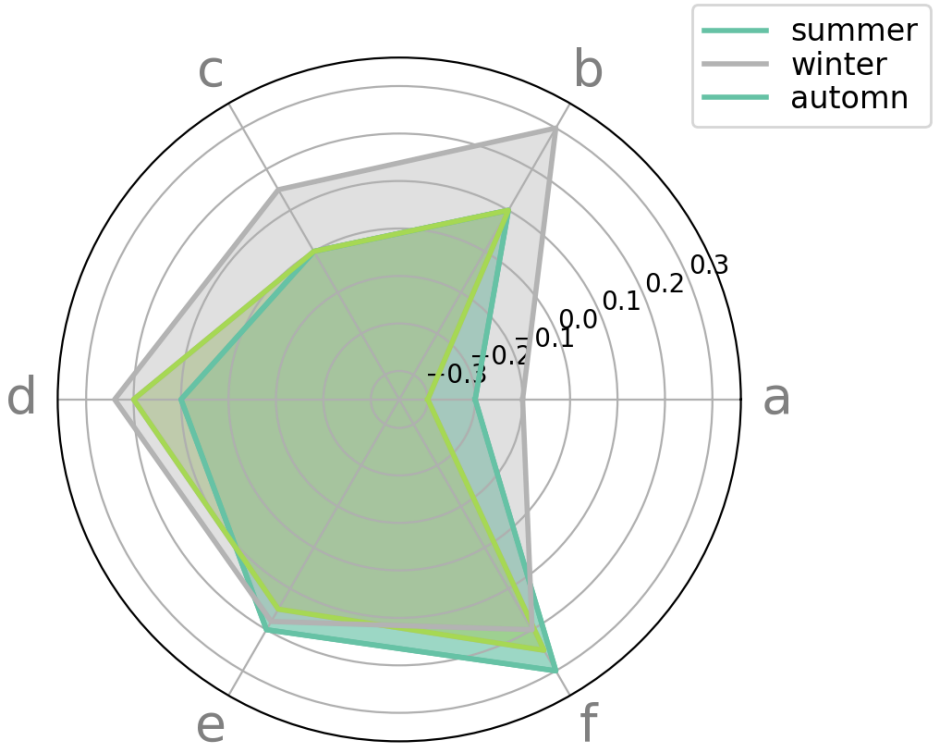
```
_ = spider_plot(values, labels, fill_kws={'color':'r'}, color='darkred')
```



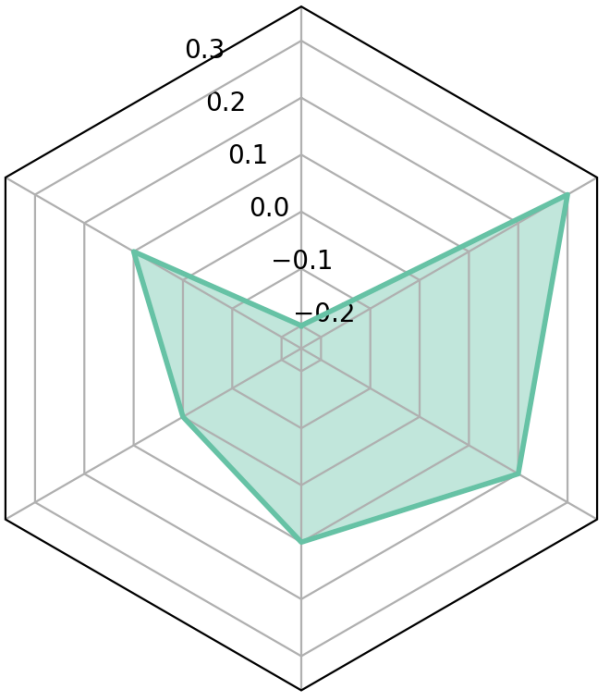
we can also specify cmap in place of color

Using dataframe

```
df = pd.DataFrame.from_dict(
    {'summer': {'a': -0.2, 'b': 0.1, 'c': 0.0, 'd': 0.1, 'e': 0.2, 'f': 0.3},
    'winter': {'a': -0.3, 'b': 0.1, 'c': 0.0, 'd': 0.2, 'e': 0.15,'f': 0.25}})

_ = spider_plot(df, xtick_kws={'size': 20})

# #%%

df = pd.DataFrame.from_dict(
    {'summer': {'a': -0.2, 'b': 0.1, 'c': 0.0, 'd': 0.1, 'e': 0.2, 'f': 0.3},
     'winter': {'a': -0.3, 'b': 0.1, 'c': 0.0, 'd': 0.2, 'e': 0.15, 'f': 0.25},
     'automn': {'a': -0.1, 'b': 0.3, 'c': 0.15, 'd': 0.24, 'e': 0.18, 'f': 0.2}})
_ = spider_plot(df, xtick_kws={'size': 20})

# #%%
# use polygon frame
_ = spider_plot(data=values, frame="polygon")
```

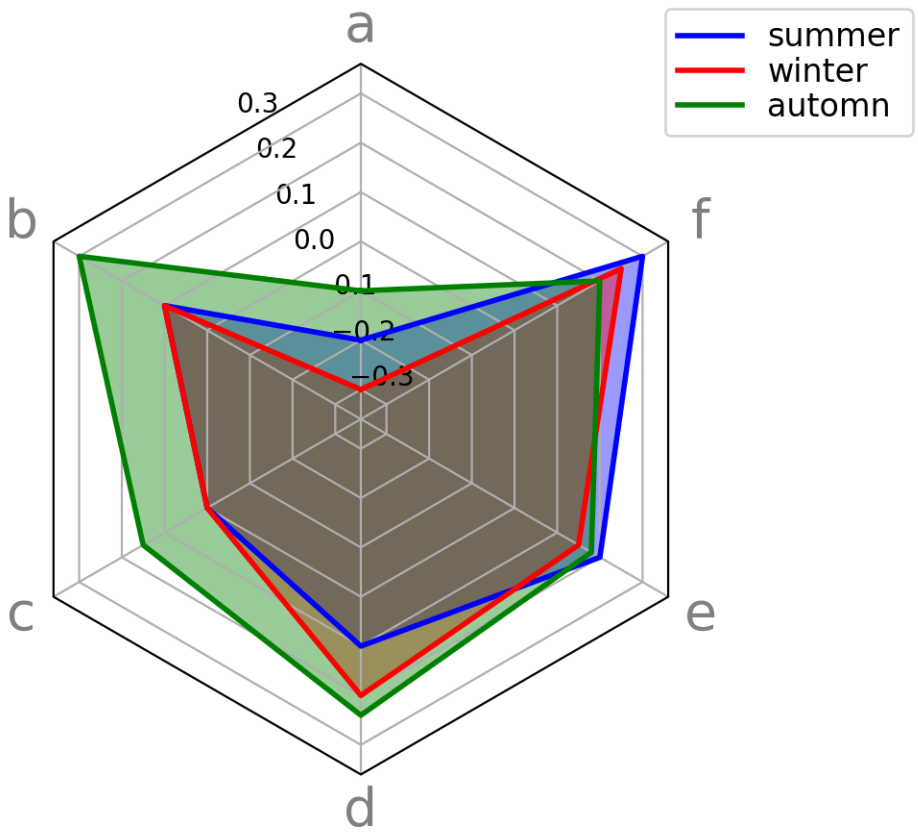

summer
winter
automn
c
b
d
a
e
f
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3

•



-

```
_ = spider_plot(df, xtick_kws={'size': 20}, frame="polygon",
           color=['b', 'r', 'g', 'm'],
            fill_color=['b', 'r', 'g', 'm'])
```



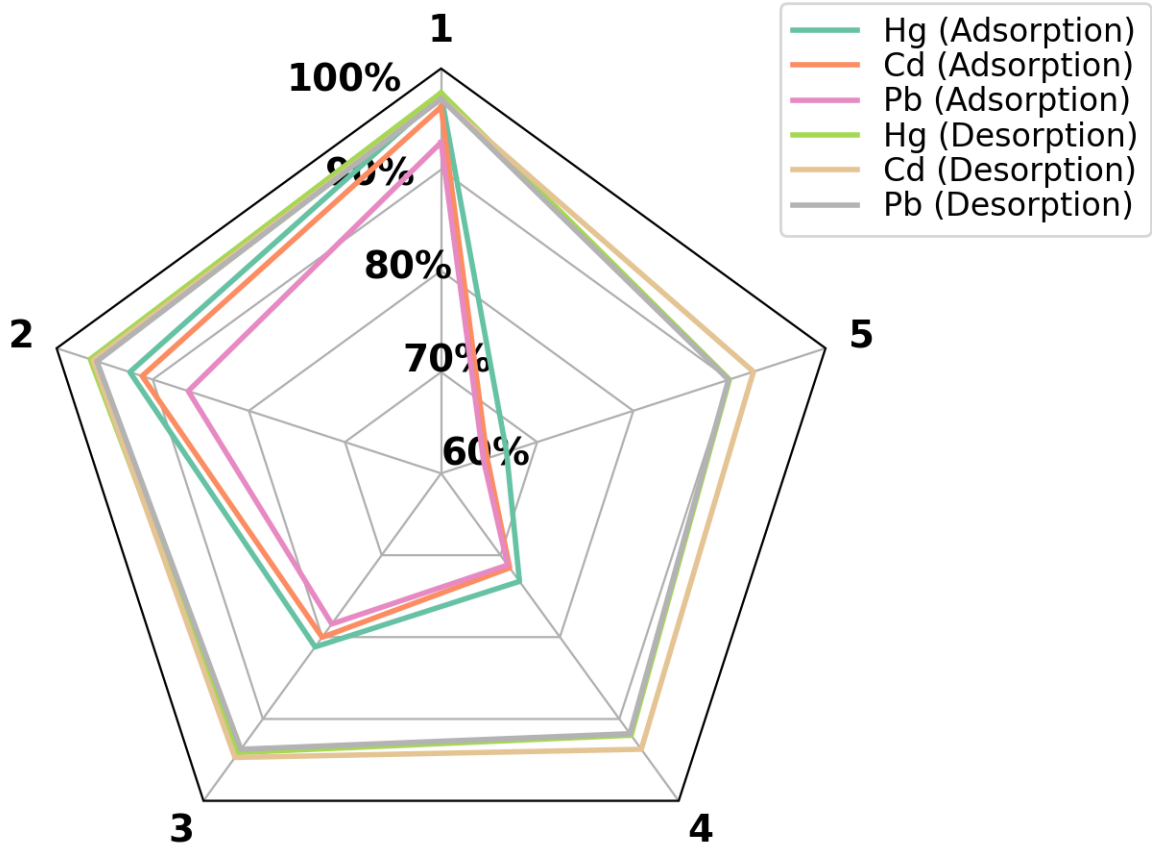
postprocessing of axes

```
df = pd.DataFrame.from_dict(
    {
        'Hg (Adsorption)': {'1': 97.4, '2': 92.38, '3': 81.2, '4': 73.2, '5': 66.81},
        'Cd (Adsorption)': {'1': 96.2, '2': 91.1, '3': 80.02, '4': 71.55, '5': 64.8},
        'Pb (Adsorption)': {'1': 92.7, '2': 86.3, '3': 78.4, '4': 71.2, '5': 64.4},
        'Hg (Desorption)': {'1': 97.6, '2': 96.5, '3': 94.1, '4': 91.99, '5': 90.0},
        'Cd (Desorption)': {'1': 97.0, '2': 96.2, '3': 94.7, '4': 93.7, '5': 92.5},
        'Pb (Desorption)': {'1': 97.0, '2': 95.8, '3': 93.7, '4': 91.8, '5': 89.9}
     })

_ = spider_plot(df, frame="polygon",
             fill_kws = {"alpha": 0.0},
             xtick_kws={'size': 14, "weight": "bold", "color": "black"},
             show=False,
             leg_kws = {'bbox_to_anchor': (0.90, 1.1)}
             )

plt.gca().set_rmax(100.0)
plt.gca().set_rmin(60.0)
plt.gca().set_rgrids((60.0, 70.0, 80.0, 90.0, 100.0),
                     ("60%", "70%", "80%", "90%", "100%"),
                     fontsize=14, weight="bold", color="black"),
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



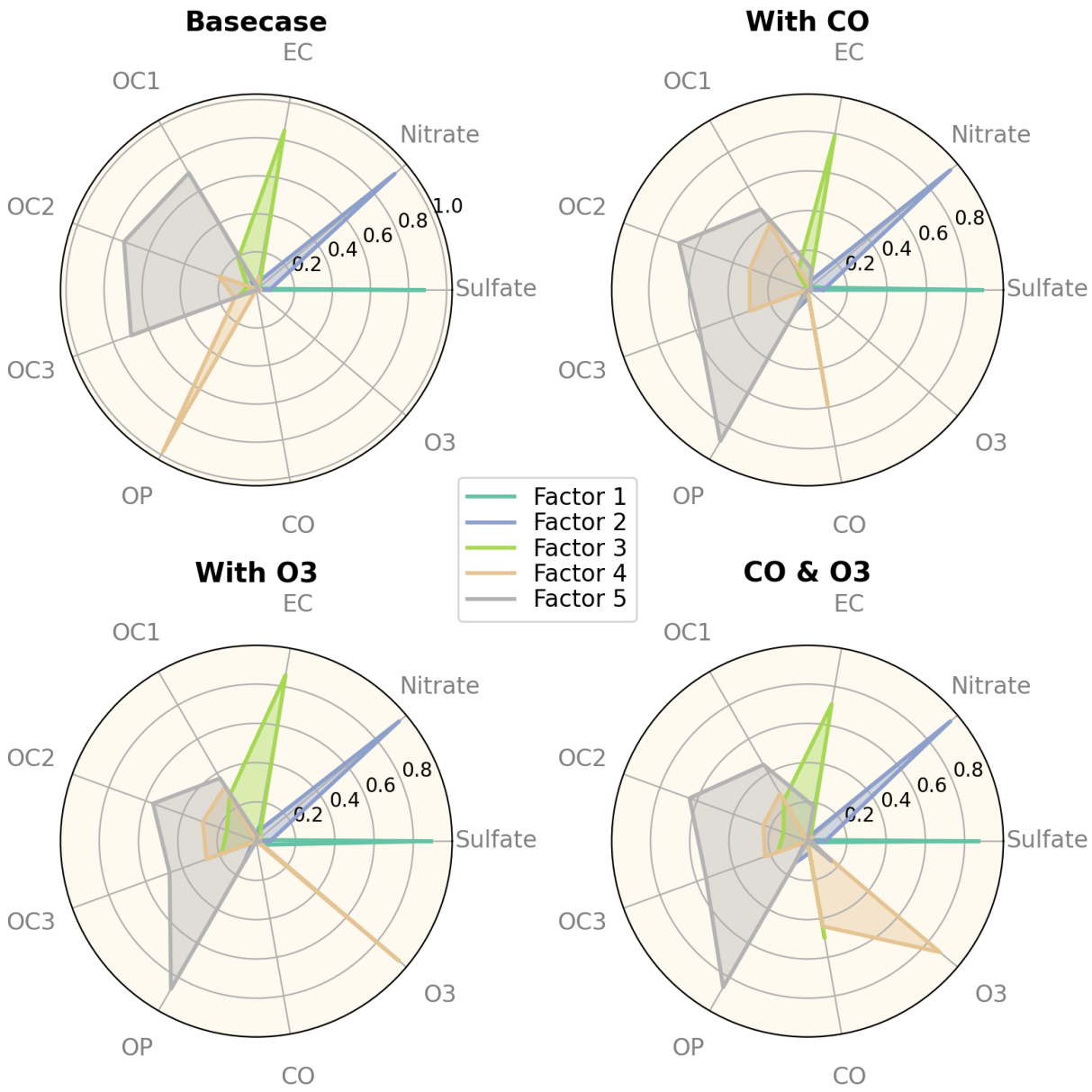
matplotlib example

```
data = [
    ['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
    ('Basecase', [
        [0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
        [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
        [0.01, 0.02, 0.85, 0.19, 0.05, 0.10, 0.00, 0.00, 0.00],
        [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
        [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]]),
    ('With CO', [
        [0.88, 0.02, 0.02, 0.02, 0.00, 0.05, 0.00, 0.05, 0.00],
        [0.08, 0.94, 0.04, 0.02, 0.00, 0.01, 0.12, 0.04, 0.00],
        [0.01, 0.01, 0.79, 0.10, 0.00, 0.05, 0.00, 0.31, 0.00],
        [0.00, 0.02, 0.03, 0.38, 0.31, 0.31, 0.00, 0.59, 0.00],
        [0.02, 0.02, 0.11, 0.47, 0.69, 0.58, 0.88, 0.00, 0.00]]),
    ('With O3', [
        [0.89, 0.01, 0.07, 0.00, 0.00, 0.05, 0.00, 0.00, 0.03],
        [0.07, 0.95, 0.05, 0.04, 0.00, 0.02, 0.12, 0.00, 0.00],
        [0.01, 0.02, 0.86, 0.27, 0.16, 0.19, 0.00, 0.00, 0.00],
```

(continues on next page)

(continued from previous page)

```
        [0.01, 0.03, 0.00, 0.32, 0.29, 0.27, 0.00, 0.00, 0.95],
        [0.02, 0.00, 0.03, 0.37, 0.56, 0.47, 0.87, 0.00, 0.00]]),
    ('CO & O3', [
        [0.87, 0.01, 0.08, 0.00, 0.00, 0.04, 0.00, 0.00, 0.01],
        [0.09, 0.95, 0.02, 0.03, 0.00, 0.01, 0.13, 0.06, 0.00],
        [0.01, 0.02, 0.71, 0.24, 0.13, 0.16, 0.00, 0.50, 0.00],
        [0.01, 0.03, 0.00, 0.28, 0.24, 0.23, 0.00, 0.44, 0.88],
        [0.02, 0.00, 0.18, 0.45, 0.64, 0.55, 0.86, 0.00, 0.16]])
]

fig, axes = create_subplots(4, subplot_kw= dict(projection='polar'),
                            figsize=(8,8))

spoke_labels = data.pop(0)
labels = ('Factor 1', 'Factor 2', 'Factor 3', 'Factor 4', 'Factor 5')

for ax, (title, col) in zip(axes.flat, data):

    d = np.array(col).transpose()
    ax.set_title(title, fontdict={'fontsize': 14, 'fontweight':'bold'})
    _ = spider_plot(d, show=False, ax=ax,
                    tick_labels=spoke_labels, xtick_kws=dict(size=12)
                    )

    ax.tick_params(axis='x', which='major', pad=12)
    ax.set_facecolor('floralwhite')

fig.legend(labels, loc='center', labelspacing=0.1, fontsize='large')
plt.tight_layout()
plt.show()
```



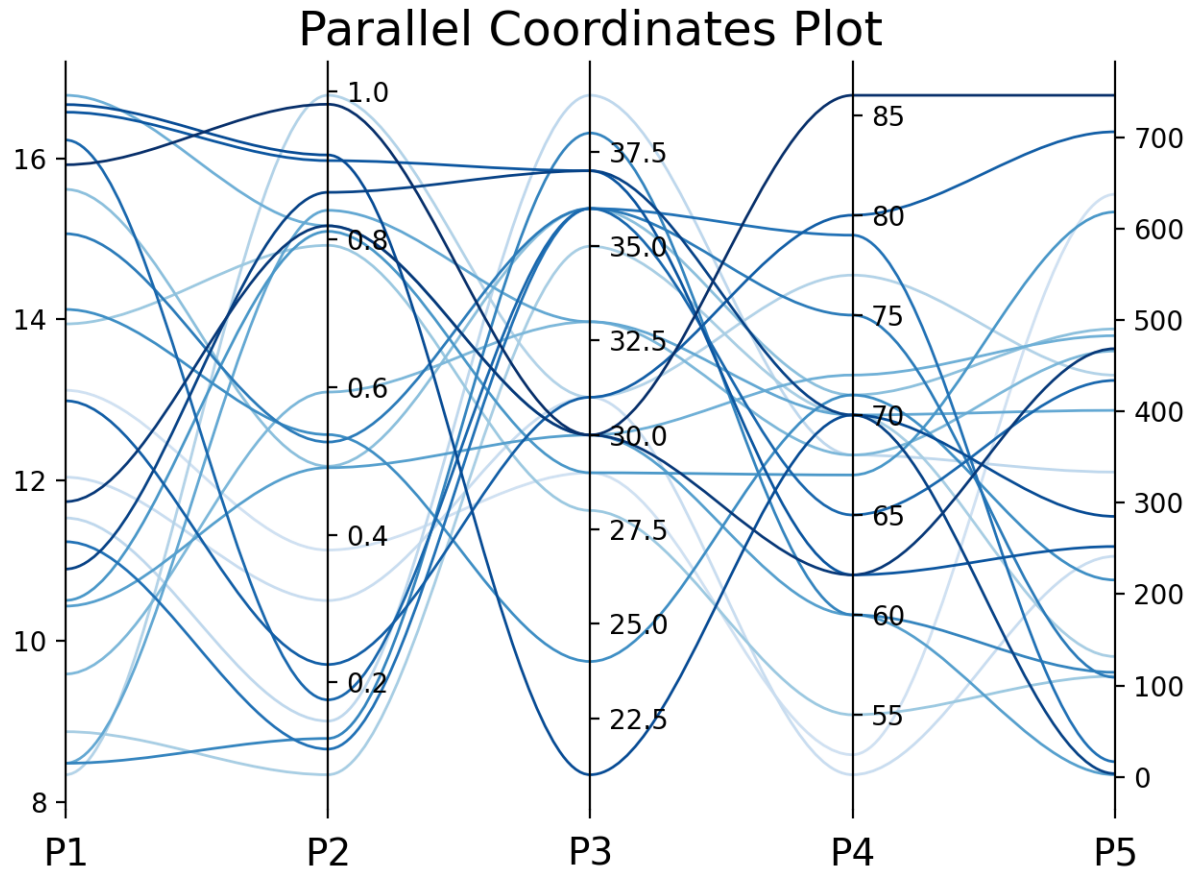
Total running time of the script: (0 minutes 5.063 seconds)

6.14 parallel_coordinates

```
import random

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from easy_mpl import parallel_coordinates
from easy_mpl.utils import version_info

version_info()
```

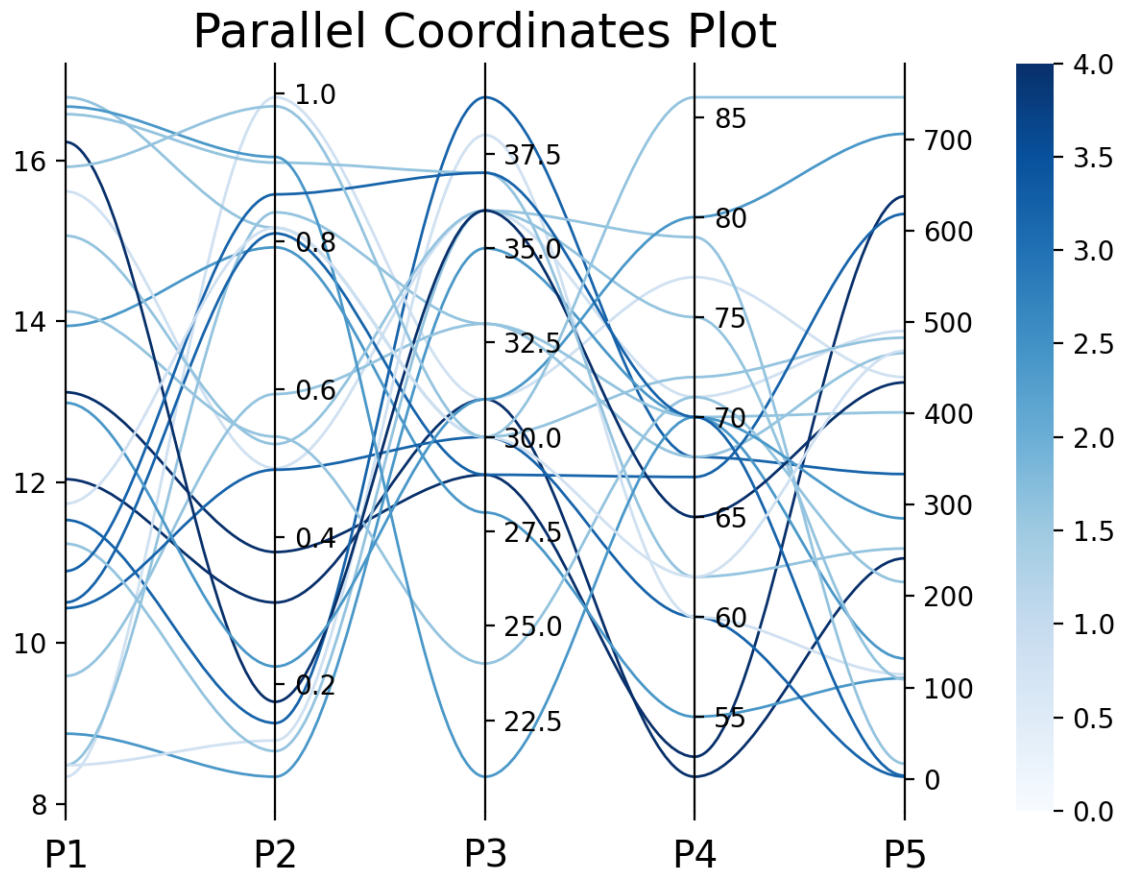
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
ynames = ['P1', 'P2', 'P3', 'P4', 'P5']  # feature/column names
N1, N2, N3 = 10, 5, 8
N = N1 + N2 + N3
categories_ = ['a', 'b', 'c', 'd', 'e', 'f']
y1 = np.random.uniform(0, 10, N) + 7
y2 = np.sin(np.random.uniform(0, np.pi, N))
y3 = np.random.binomial(300, 1 / 10, N)
y4 = np.random.binomial(200, 1 / 3, N)
y5 = np.random.uniform(0, 800, N)
# combine all arrays into a pandas DataFrame
data_np = np.column_stack((y1, y2, y3, y4, y5))
data_df = pd.DataFrame(data_np, columns=ynames)
# using a DataFrame to draw parallel coordinates
_ = parallel_coordinates(data_df, names=ynames)
```



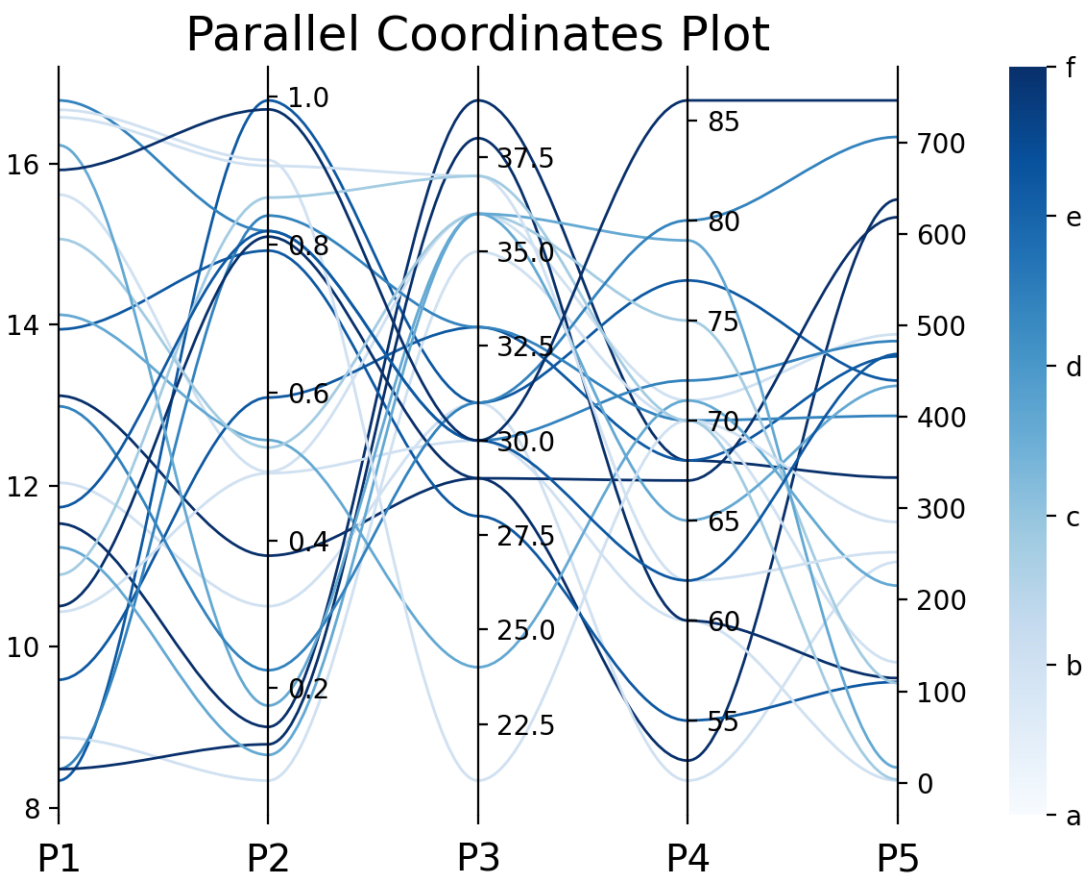
using continuous values for categories

```
_ = parallel_coordinates(data_df, names=ynames, categories=np.random.randint(0, 5, N))
```



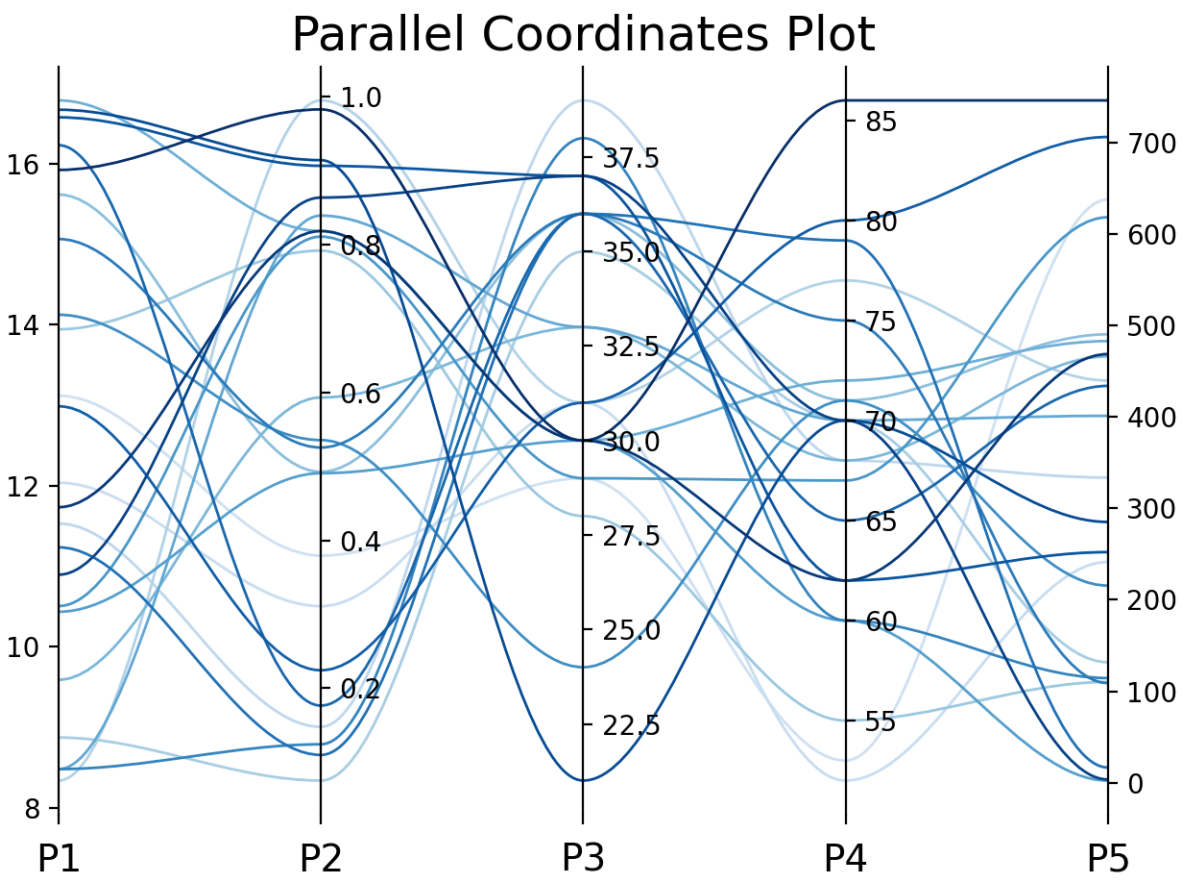
using categorical classes

```
_ = parallel_coordinates(data_df, names=ynames, categories=random.choices(categories_,␣
↪k=N))
```



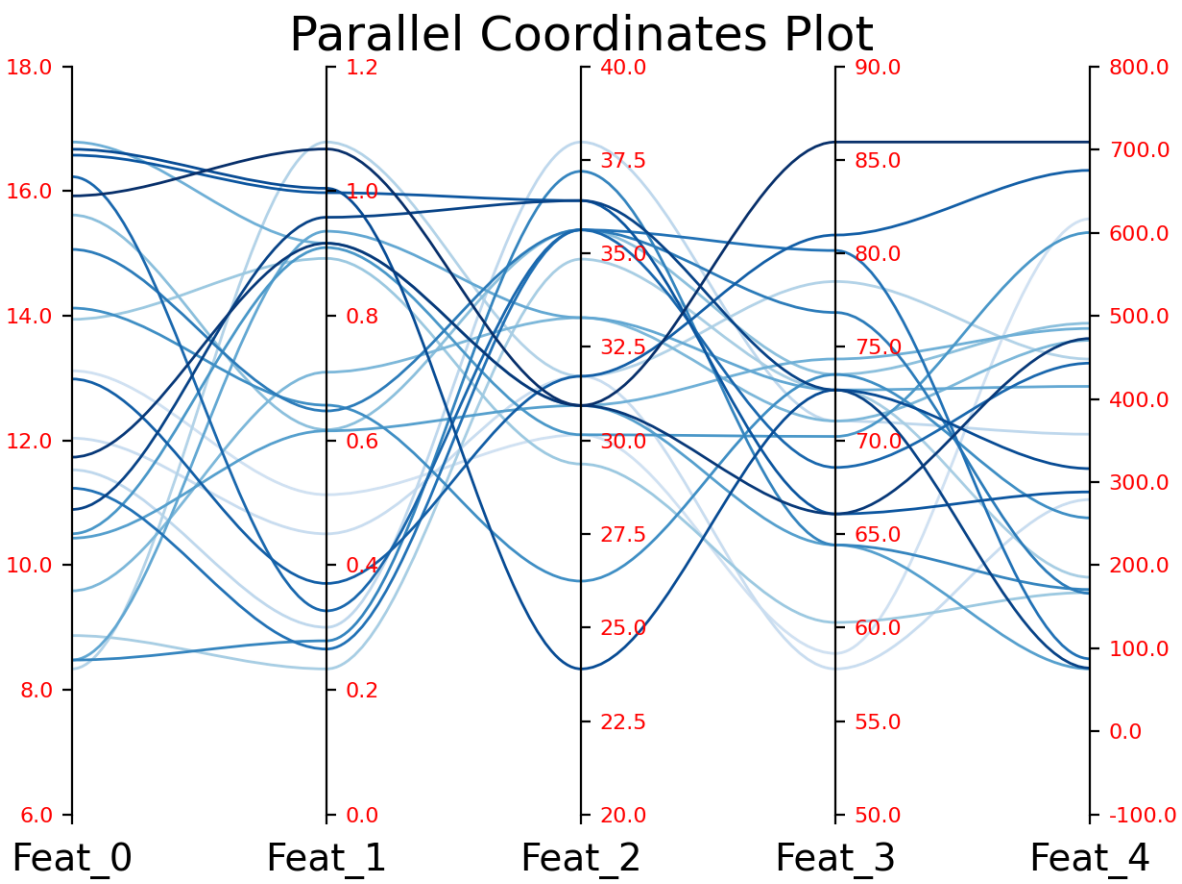
using numpy array instead of DataFrame

```
_ = parallel_coordinates(data_df.values, names=ynames)
```



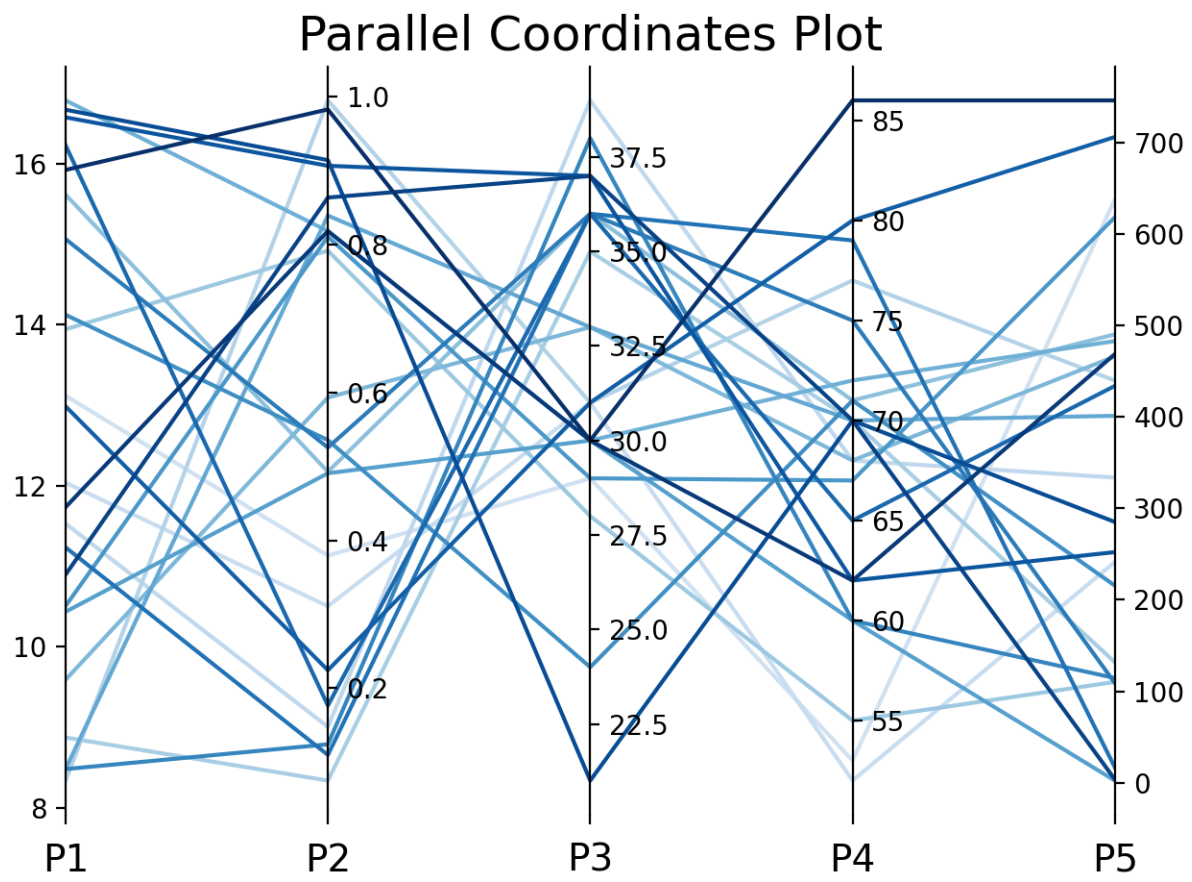
with customized tick labels

```
_ = parallel_coordinates(data_df.values, ticklabel_kws={"fontsize": 8, "color": "red"})
```



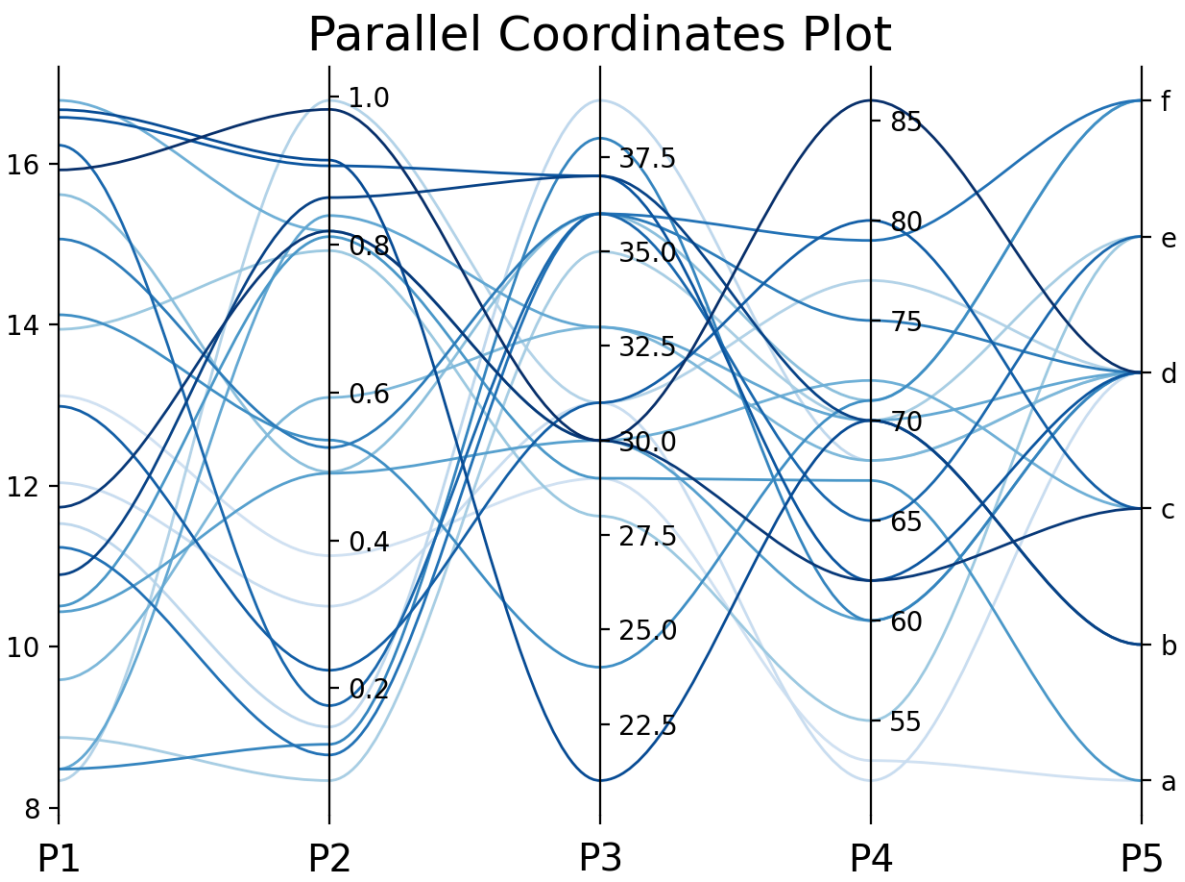
using straight lines instead of bezier

```
_ = parallel_coordinates(data_df, linestyle="straight")
```



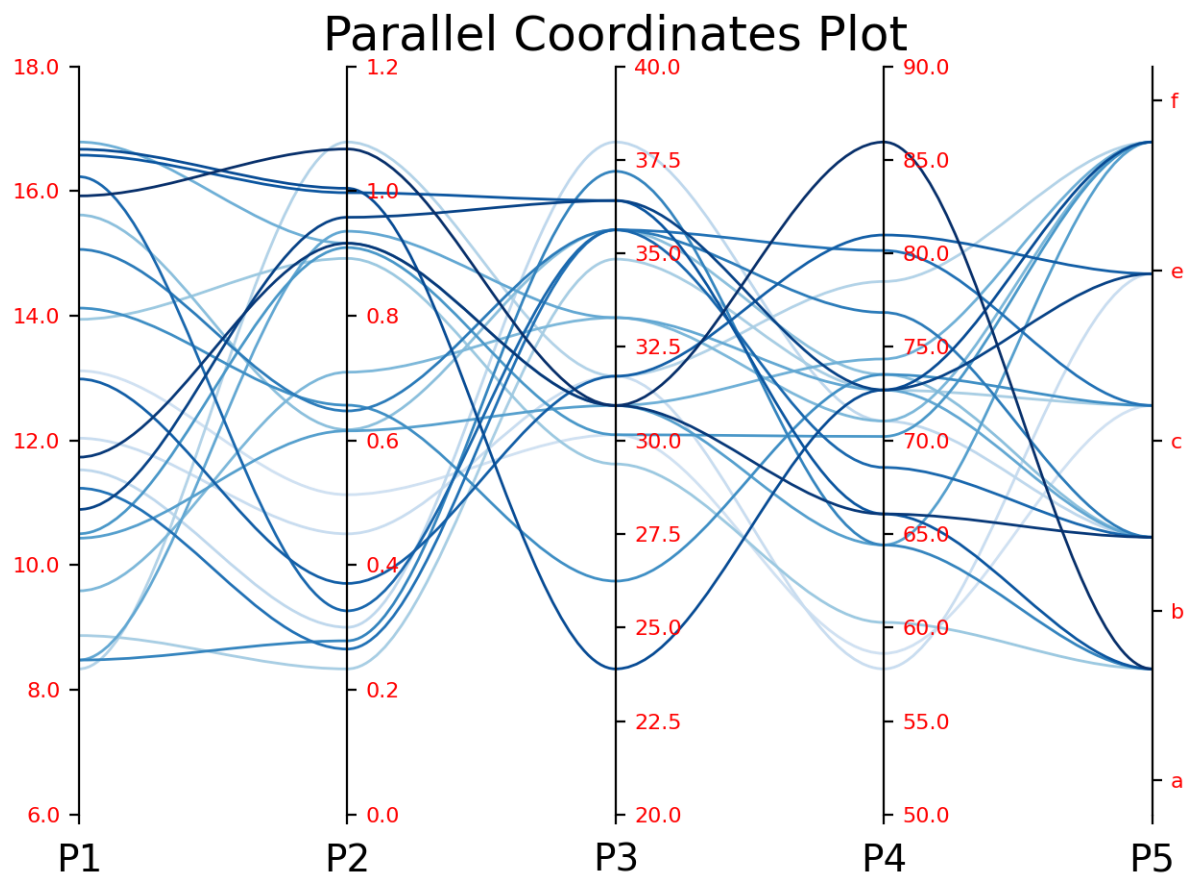
with categorical class labels

```
data_df['P5'] = random.choices(categories_, k=N)
_ = parallel_coordinates(data_df, names=ynames)
```



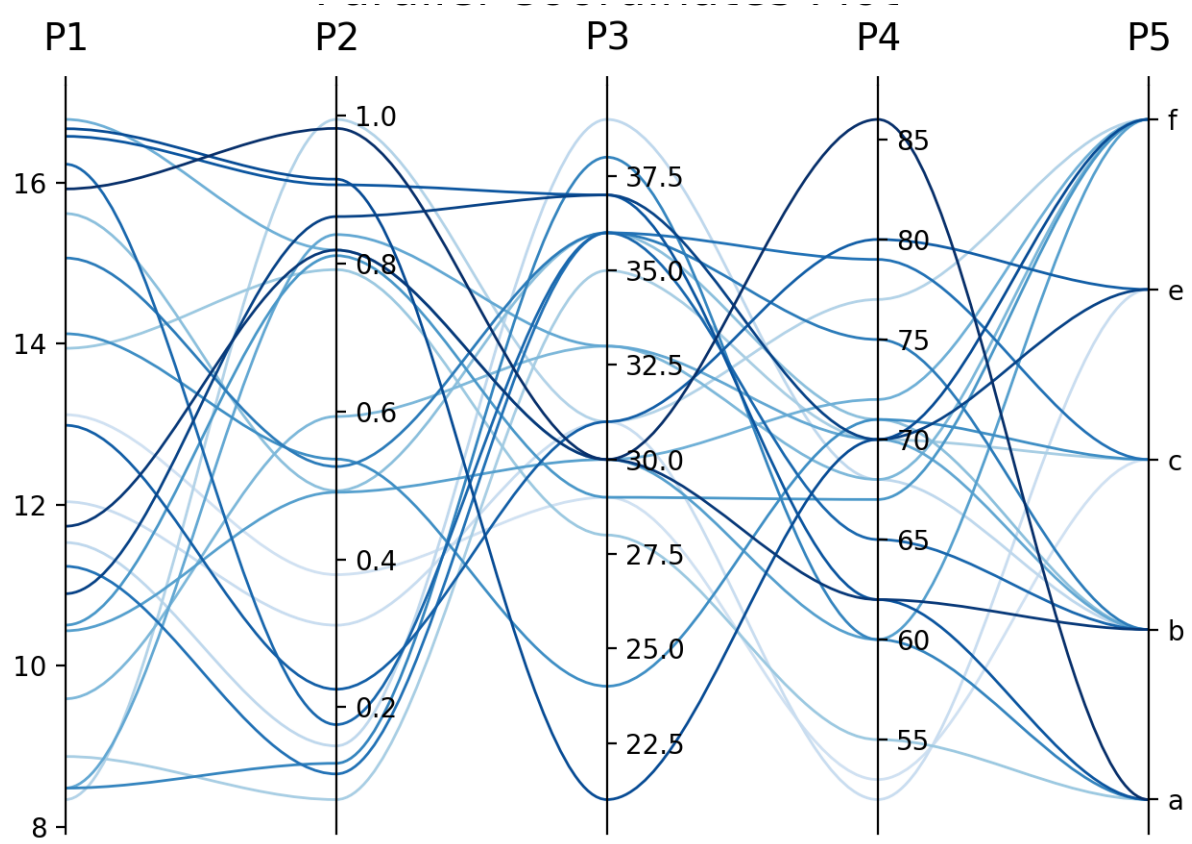
with categorical class labels and customized ticklabels

```
data_df['P5'] = random.choices(categories_, k=N)
_ = parallel_coordinates(data_df,  ticklabel_kws={"fontsize": 8, "color": "red"})
```



show parameter labels at the top

```
axes = parallel_coordinates(data_df, show=False)
axes.xaxis.tick_top()
plt.show()
```



Total running time of the script: (0 minutes 7.376 seconds)

6.15 taylor plot

```
import numpy as np
from easy_mpl import taylor_plot
from easy_mpl.utils import version_info

version_info()

# sphinx_gallery_thumbnail_number = -1
```

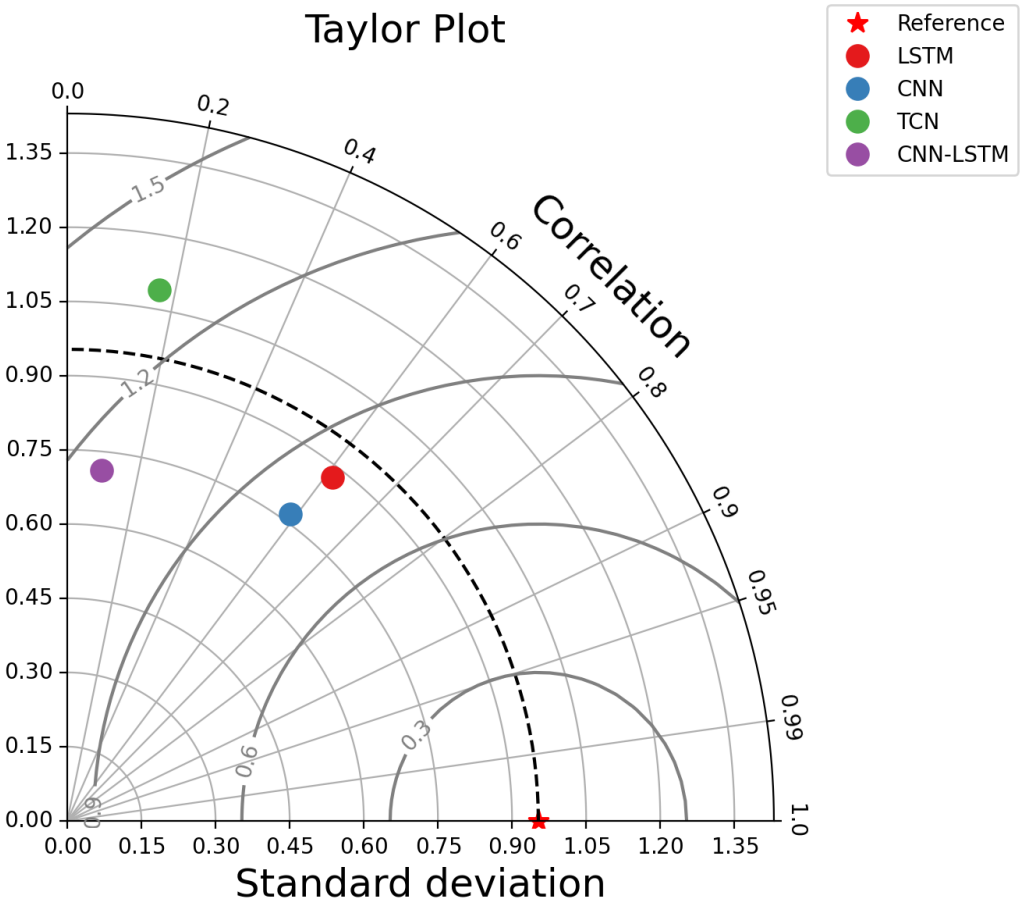
```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
# The desired covariance matrix.
cov = np.array(
       [[1,  0.8, 0.6, 0.4, 0.2],
       [0.8, 1.2, 0.8, 0.6, 0.4],
       [0.6, 0.8, 0.8, 0.8, 0.6],
       [0.4, 0.6, 0.8, 1.4, 0.8],
       [0.2, 0.4, 0.6, 0.8, 0.6]]
```

(continues on next page)

(continued from previous page)

```
)

# Generate the random samples.
rng = np.random.default_rng(313)
data = rng.multivariate_normal(np.zeros(5), cov, size=100)
print(data.shape)

observations =  data[:, 0]
simulations =  {"LSTM": data[:, 1],
            "CNN": data[:, 2],
            "TCN": data[:, 3],
            "CNN-LSTM": data[:, 4]}
_ = taylor_plot(observations=observations,
            simulations=simulations,
            title="Taylor Plot")
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_taylor.py:29: RuntimeWarning: covariance is not symmetric positive-
↪semidefinite.
  data = rng.multivariate_normal(np.zeros(5), cov, size=100)
(100, 5)
```

multiple taylor plots in one figure

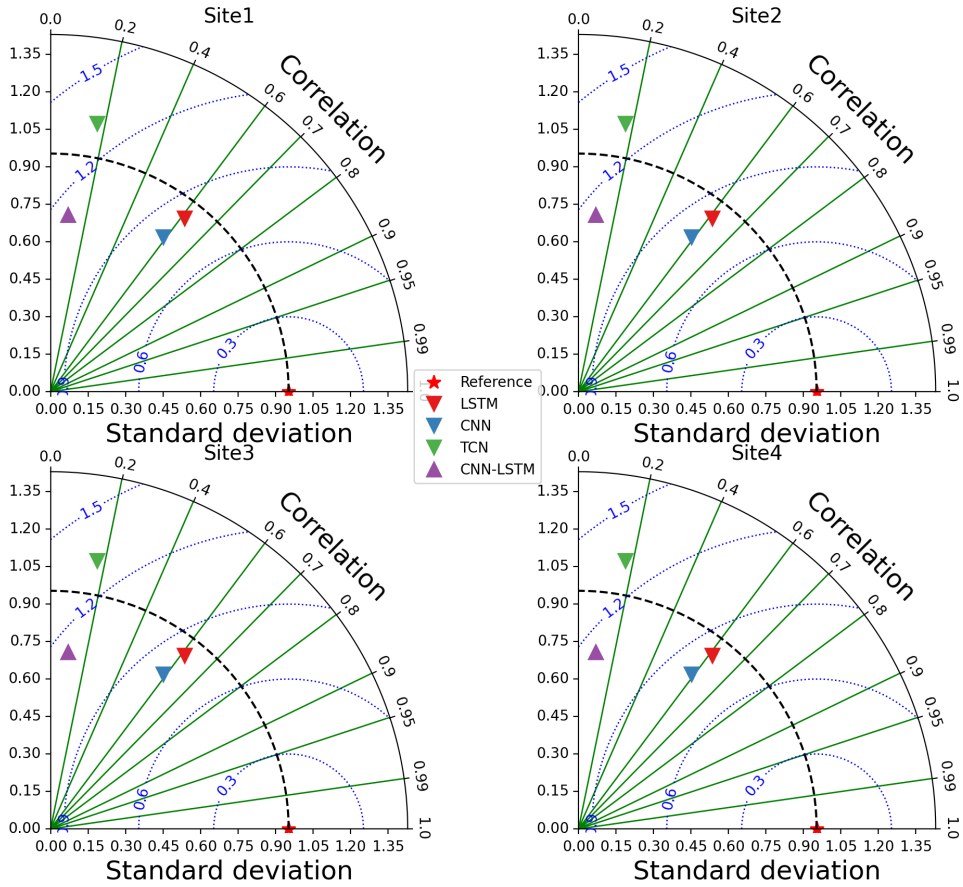
```
def create_data(cov, seed=313, mu=np.zeros(5), size=100):

    # Generate the random samples.
    rng = np.random.default_rng(seed)

    return rng.multivariate_normal(np.zeros(5), cov, size=size)

cov1 = np.array(
    [[1,  0.8, 0.6, 0.4, 0.2],
    [0.8, 1.2, 0.8, 0.6, 0.4],
    [0.6, 0.8, 0.8, 0.8, 0.6],
    [0.4, 0.6, 0.8, 1.4, 0.8],
    [0.2, 0.4, 0.6, 0.8, 0.6]]
)

cov2 = np.array(
    [[1,  0.8, 0.6, 0.4, 0.2],
    [0.8, 1.2, 0.8, 0.6, 0.4],
    [0.6, 0.8, 0.8, 0.8, 0.6],
    [0.4, 0.6, 0.8, 1.4, 0.8],
    [0.2, 0.4, 0.6, 0.8, 0.6]]
)

cov3 = np.array(
    [[1,  0.8, 0.6, 0.4, 0.2],
    [0.8, 1.2, 0.8, 0.6, 0.4],
    [0.6, 0.8, 0.8, 0.8, 0.6],
    [0.4, 0.6, 0.8, 1.4, 0.8],
    [0.2, 0.4, 0.6, 0.8, 0.6]]
)

cov4 = np.array(
    [[1,  0.8, 0.6, 0.4, 0.2],
    [0.8, 1.2, 0.8, 0.6, 0.4],
    [0.6, 0.8, 0.8, 0.8, 0.6],
    [0.4, 0.6, 0.8, 1.4, 0.8],
    [0.2, 0.4, 0.6, 0.8, 0.6]]
)

site1_data = create_data(cov1)
site2_data = create_data(cov2)
site3_data = create_data(cov3)
site4_data = create_data(cov4)

observations = {
    'site1': site1_data[:, 0],
    'site2': site2_data[:, 0],
    'site3': site3_data[:, 0],
    'site4': site4_data[:, 0],
}
```

(continues on next page)

(continued from previous page)

```
simulations = {
    "site1": {"LSTM": site1_data[:, 1],
              "CNN": site1_data[:, 2],
              "TCN": site1_data[:, 3],
              "CNN-LSTM": site1_data[:, 4]},

    "site2": {"LSTM": site2_data[:, 1],
              "CNN": site2_data[:, 2],
              "TCN": site2_data[:, 3],
              "CNN-LSTM": site2_data[:, 4]},

    "site3": {"LSTM": site3_data[:, 1],
              "CNN": site3_data[:, 2],
              "TCN": site3_data[:, 3],
              "CNN-LSTM": site3_data[:, 4]},

    "site4": {"LSTM": site4_data[:, 1],
              "CNN": site4_data[:, 2],
              "TCN": site4_data[:, 3],
              "CNN-LSTM": site4_data[:, 4]},
}

# define positions of subplots

rects = dict(site1=221, site2=222, site3=223, site4=224)

_ = taylor_plot(observations=observations,
            simulations=simulations,
            axis_locs=rects,
            plot_bias=True,
            cont_kws={'colors': 'blue', 'linewidths': 1.0, 'linestyles': 'dotted'},
            grid_kws={'axis': 'x', 'color': 'g', 'lw': 1.0},
            title="mutiple subplots")
```

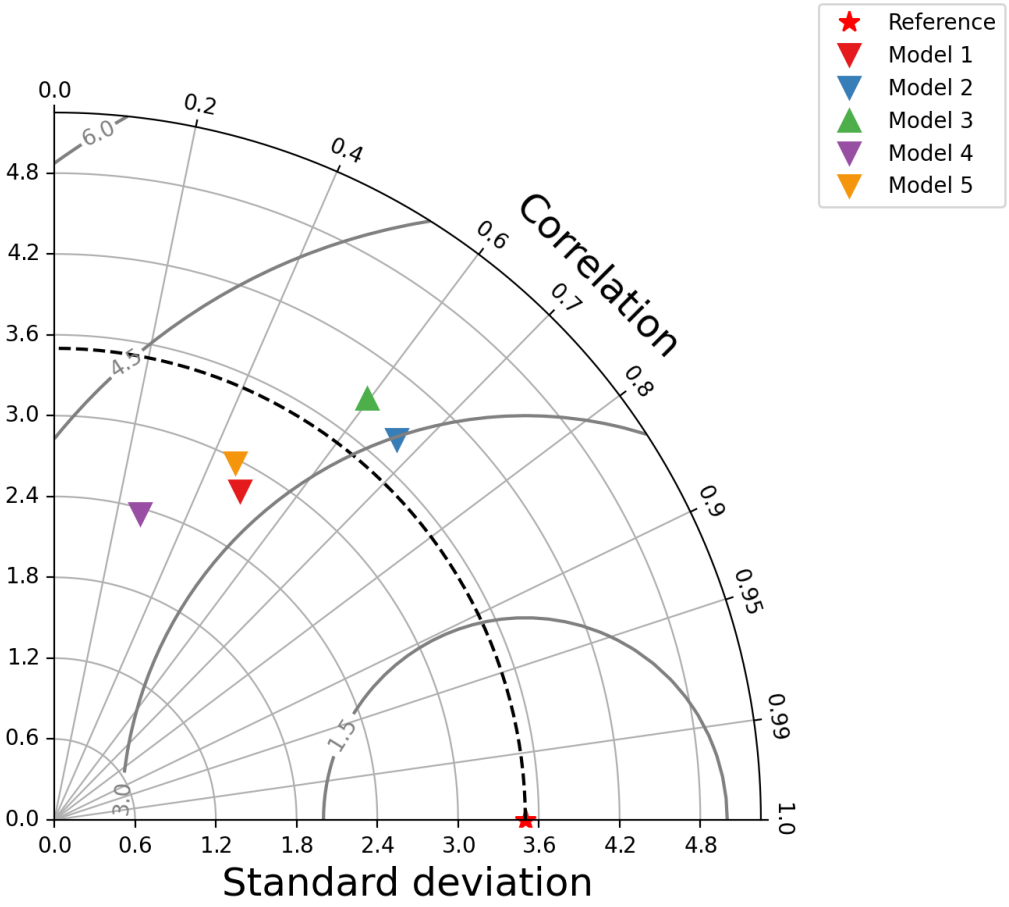
mutiple subplots




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_taylor.py:49: RuntimeWarning: covariance is not symmetric positive-
↪semidefinite.
  return rng.multivariate_normal(np.zeros(5), cov, size=size)
```

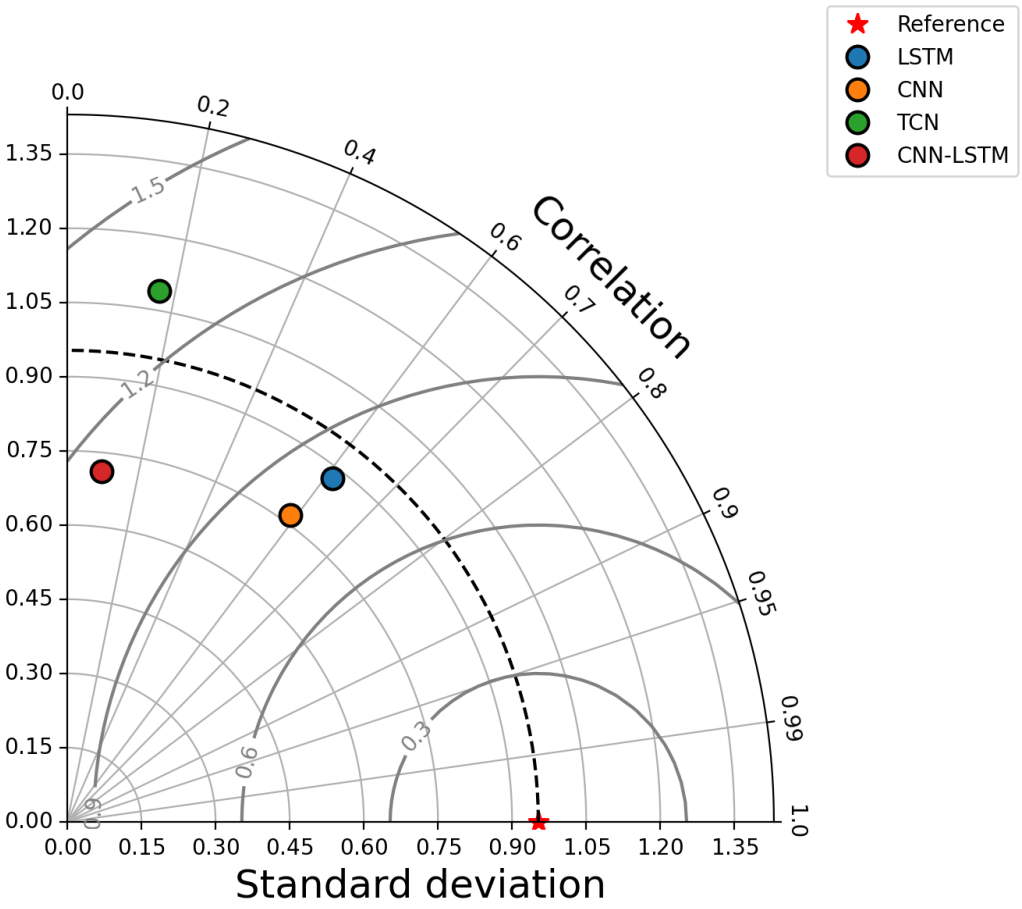
using statistics instead of arrays

```
observations = {'std': 3.5}
predictions = {   # pbias is optional
        'Model 1': {'std': 2.80068, 'corr_coeff': 0.49172, 'pbias': -8.85},
        'Model 2': {'std': 3.8, 'corr_coeff': 0.67, 'pbias': -19.76},
        'Model 3': {'std': 3.9, 'corr_coeff': 0.596, 'pbias': 7.81},
        'Model 4': {'std': 2.36, 'corr_coeff': 0.27, 'pbias': -22.78},
        'Model 5': {'std': 2.97, 'corr_coeff': 0.452, 'pbias': -7.99}}



_ = taylor_plot(observations,
    predictions)
```



with customized markers

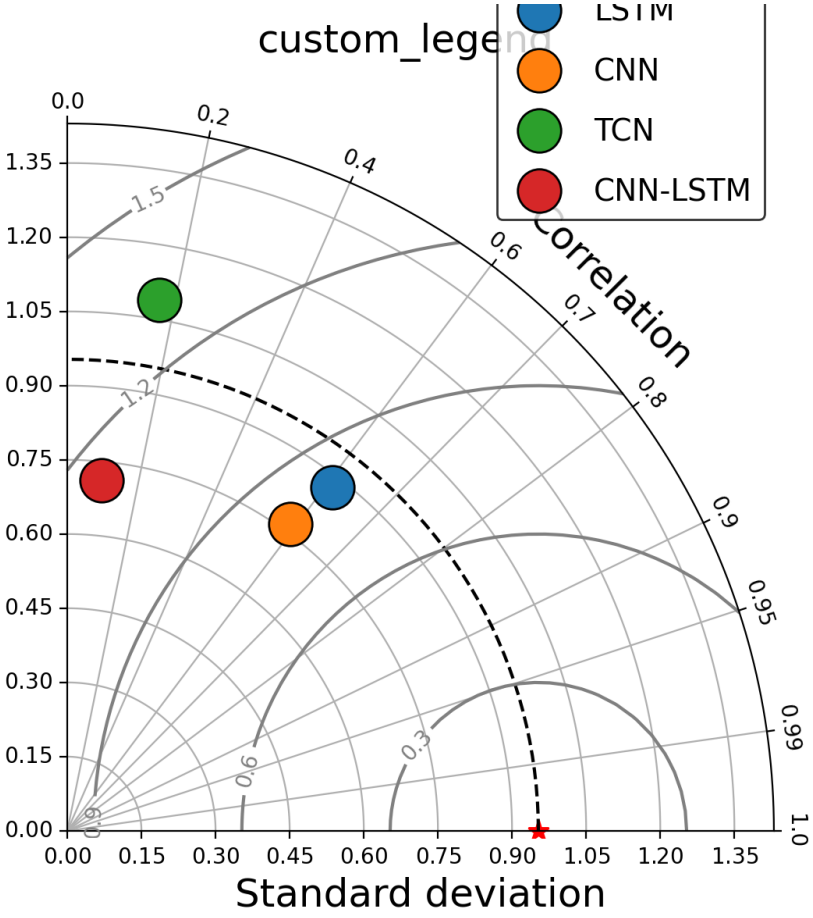
```
cov = np.array(
    [[1, 0.8, 0.6, 0.4, 0.2],
     [0.8, 1.2, 0.8, 0.6, 0.4],
     [0.6, 0.8, 0.8, 0.8, 0.6],
     [0.4, 0.6, 0.8, 1.4, 0.8],
     [0.2, 0.4, 0.6, 0.8, 0.6]]
)

data = create_data(cov)

observations = data[:, 0]
simulations = {"LSTM": data[:, 1],
               "CNN": data[:, 2],
               "TCN": data[:, 3],
               "CNN-LSTM": data[:, 4]}
_ = taylor_plot(observations=observations,
                simulations=simulations,
                marker_kws={'markersize': 10, 'markeredgewidth': 1.5,
                            'markeredgecolor': 'black', 'lw': 0.0})
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/checkouts/latest/scripts/
↪plotting/_taylor.py:49: RuntimeWarning: covariance is not symmetric positive-
↪semidefinite.
  return rng.multivariate_normal(np.zeros(5), cov, size=size)
```

with customizing bbox

```
_ = taylor_plot(observations=observations,
            simulations=simulations,
            title="custom_legend",
            leg_kws={'facecolor': 'white',
                'edgecolor': 'black','bbox_to_anchor':(0.80, 1.1),
                'fontsize': 14, 'labelspacing': 1.0},
                marker_kws = {'ms':'20', 'markeredgecolor': 'k', 'lw': 0.0},
                )
```



Total running time of the script: (0 minutes 3.348 seconds)

6.16 boxplot

This file shows the usage of `boxplot()` function.

```
# sphinx_gallery_thumbnail_number = -3

import pandas as pd
import matplotlib.pyplot as plt

from easy_mpl import boxplot, plot
from easy_mpl.utils import _rescale
from easy_mpl.utils import version_info

version_info()  # print version information of all the packages being used
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
f = "https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/datasets/arg_
↪busan.csv"
```

(continues on next page)

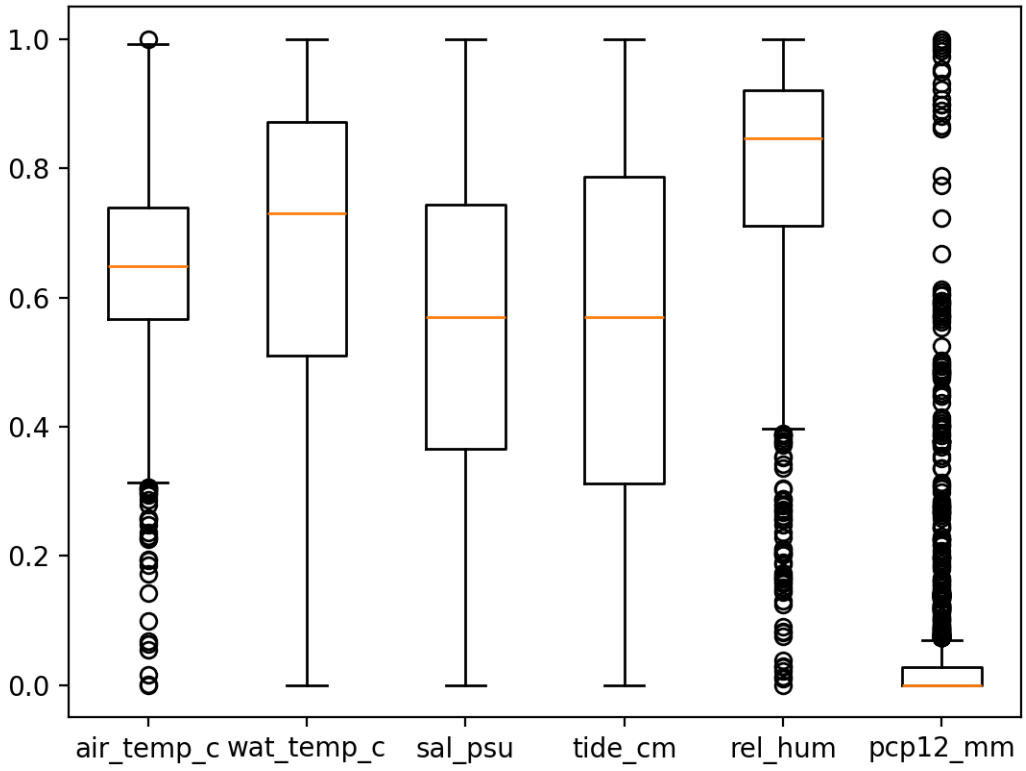
(continued from previous page)

```
dataframe = pd.read_csv(f, index_col='index')
cols = ['air_temp_c', 'wat_temp_c', 'sal_psu', 'tide_cm', 'rel_hum', 'pcp12_mm']
df = dataframe.copy()
for col in df.columns:
    df[col] = _rescale(df[col].values)

print(f"Our data has {len(df)} rows and {df.shape[1]} columns")
```

```
Our data has 1446 rows and 25 columns
```

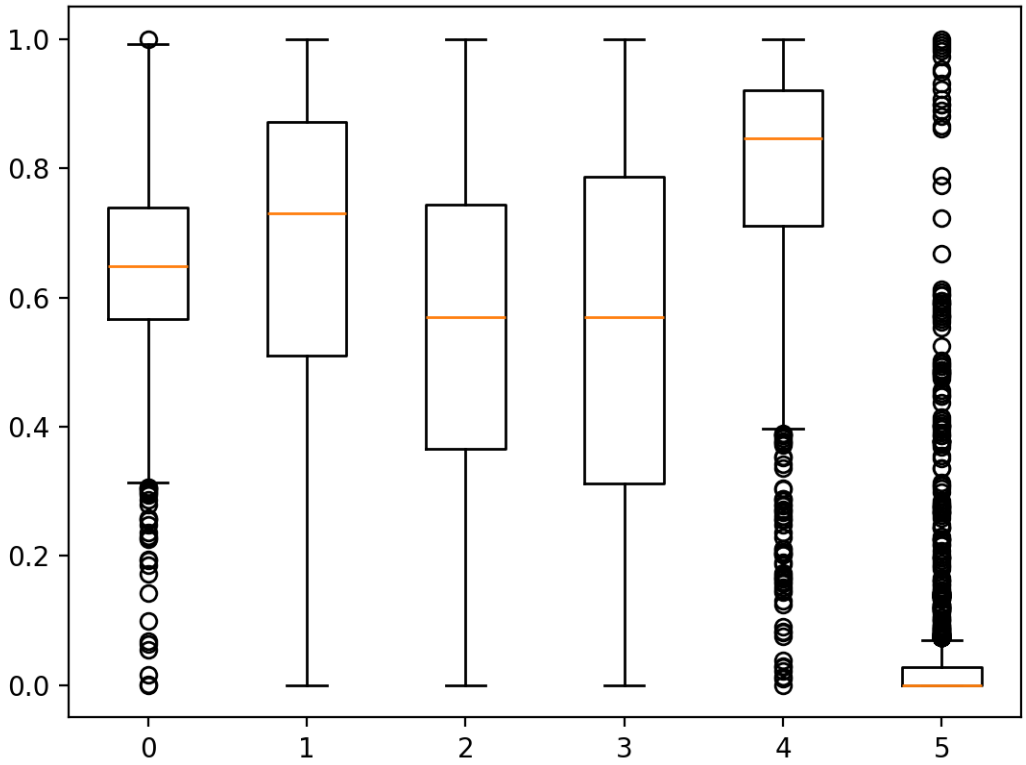
To draw a boxplot we can provide a pandas DataFrame

```
_ = boxplot(df[cols], fill_color='khaki')
```



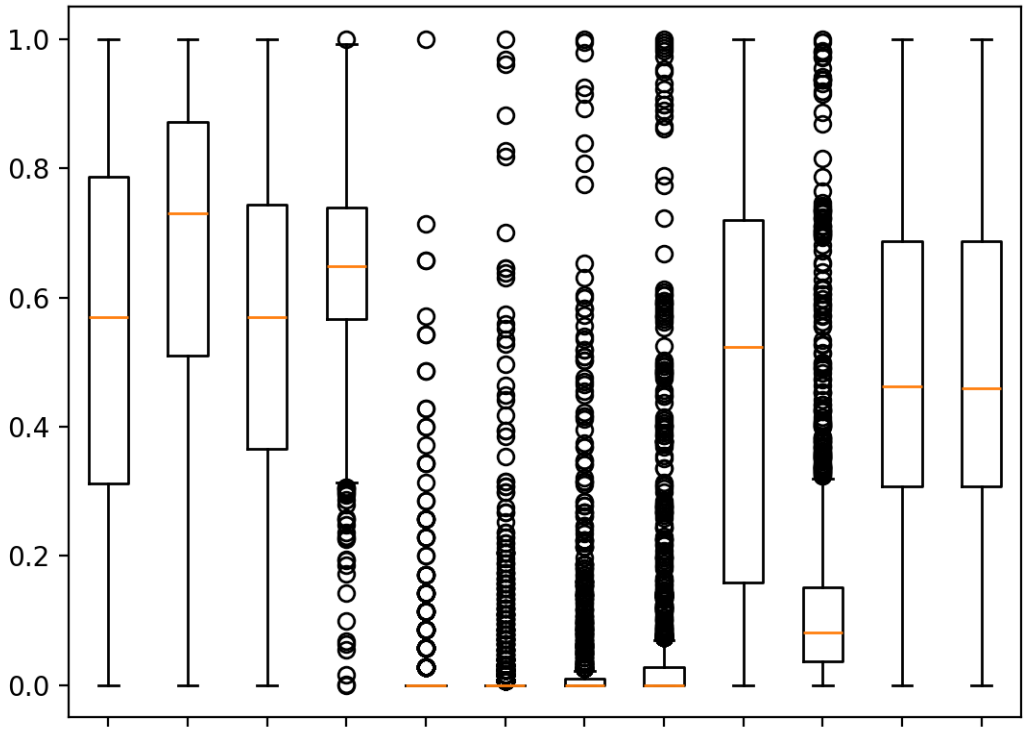
We can also provide multiple (numpy) array

```
_ = boxplot(df[cols].values)
```



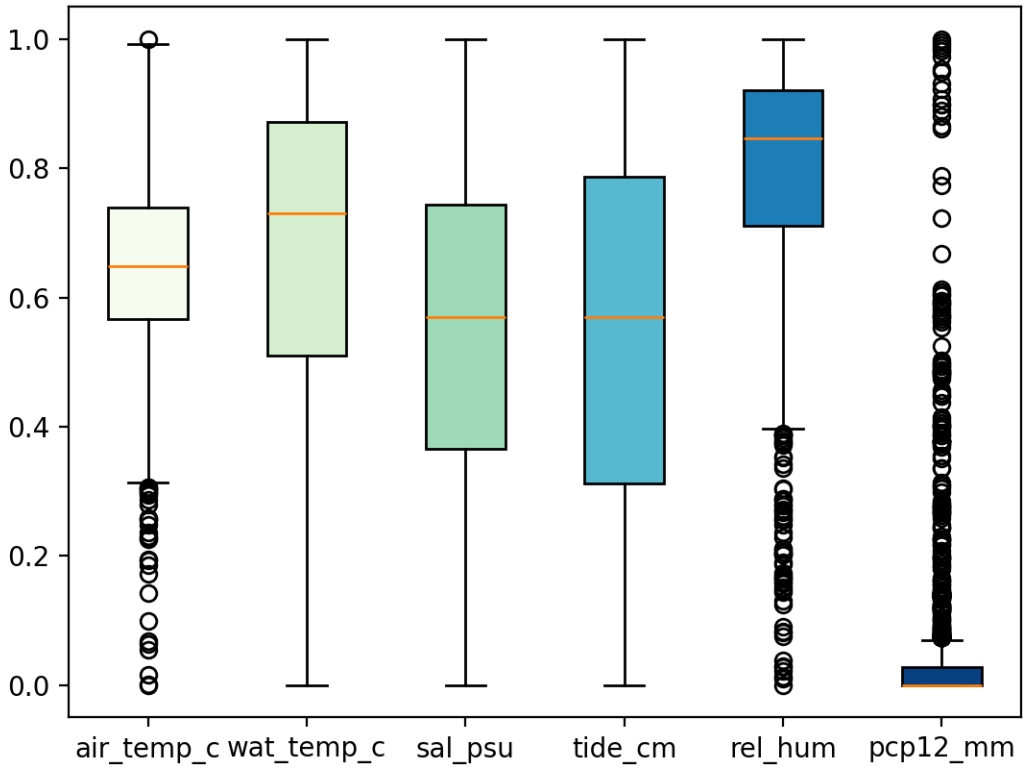
We can give the list of arrays

```
data = df.iloc[:, 0:12]
_ = boxplot([data[col].values for col in data.columns])
```

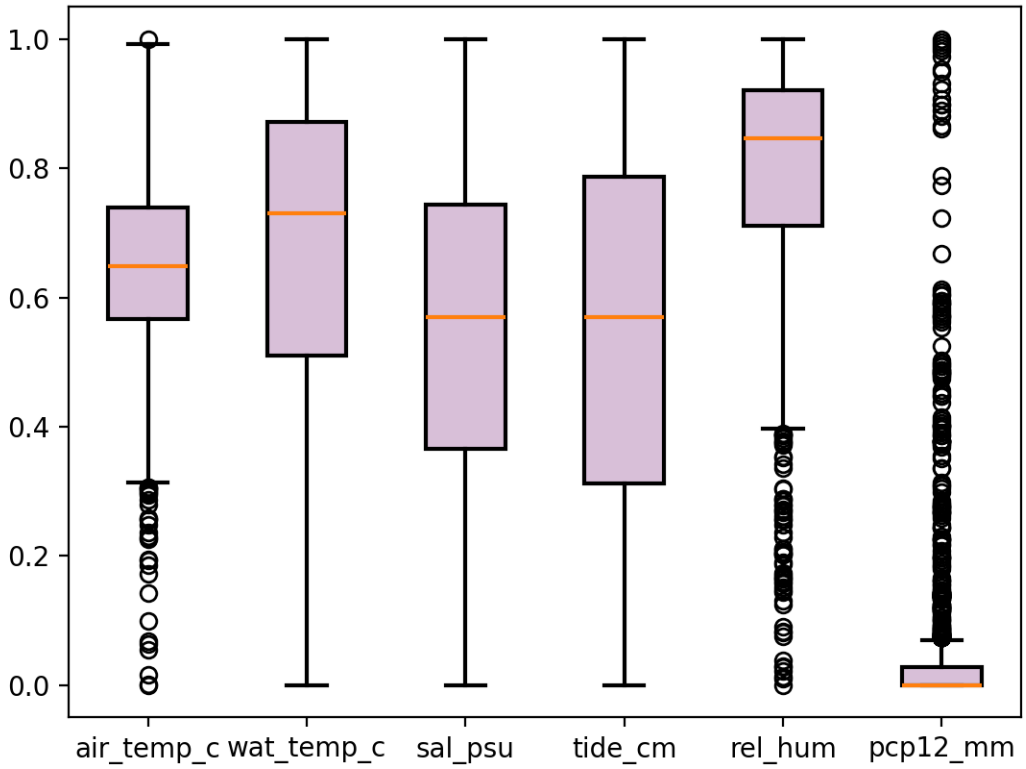


The fill color can be specificed using any valid matplotlib cmap

```
_ = boxplot(df[cols], fill_color="GnBu", patch_artist=True)
```

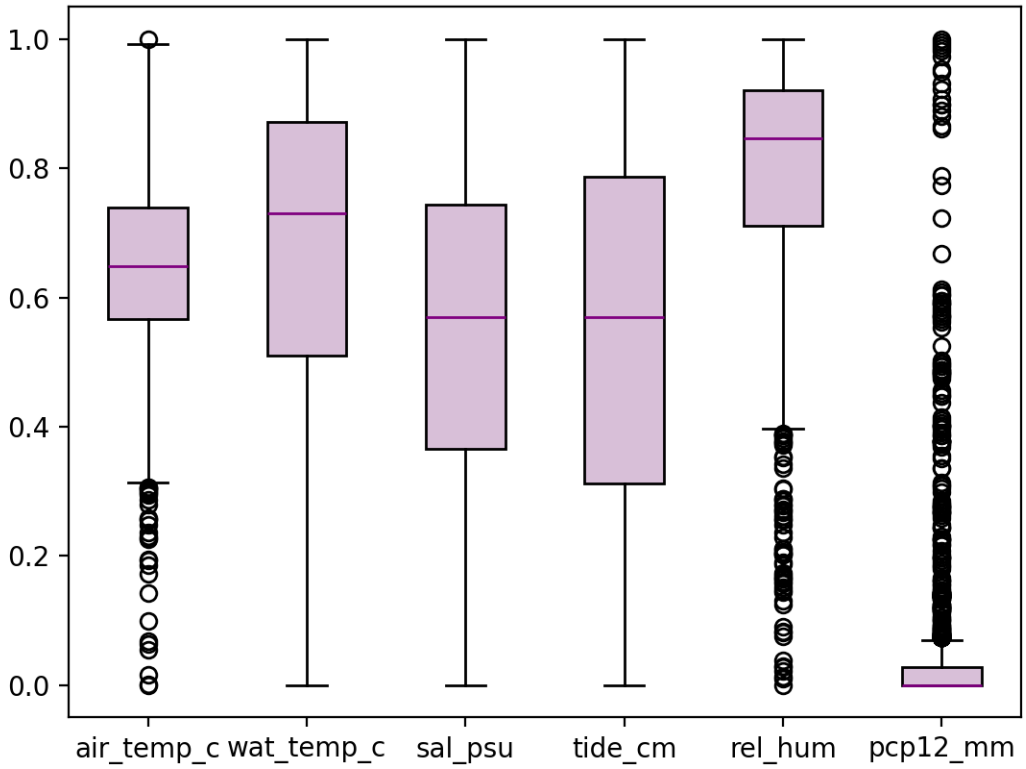



```
_ = boxplot(df[cols], fill_color="thistle", line_width=1.5, patch_artist=True)
```



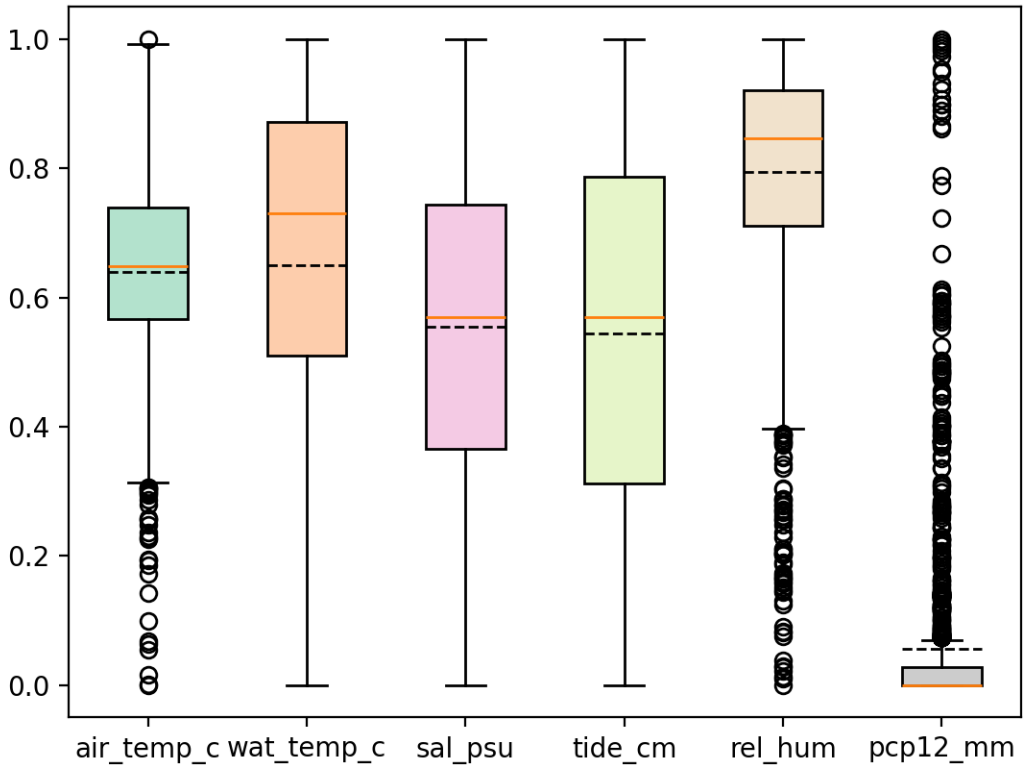
change color of median line

```
_ = boxplot(df[cols], fill_color="thistle", patch_artist=True,
            medianprops={"color": "purple"})
```



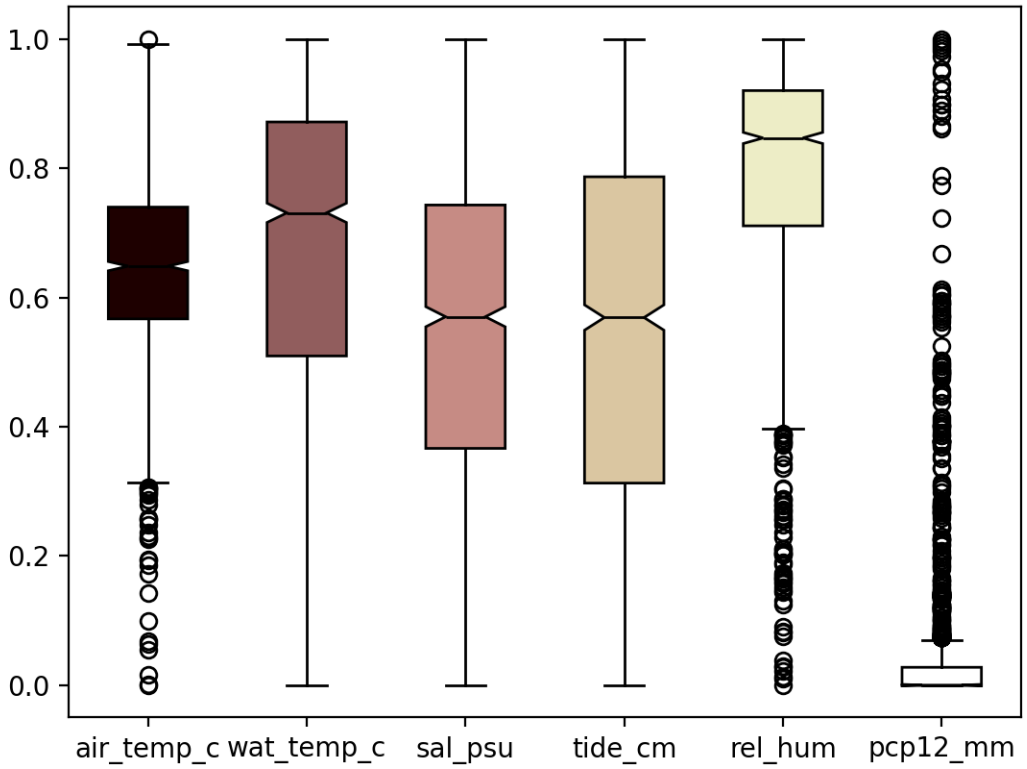
different color for box boundary and filling the box

```
_ = boxplot(df[cols], fill_color="Pastel2", patch_artist=True,
            meanline=True, showmeans=True, meanprops={"color": "black"})
```



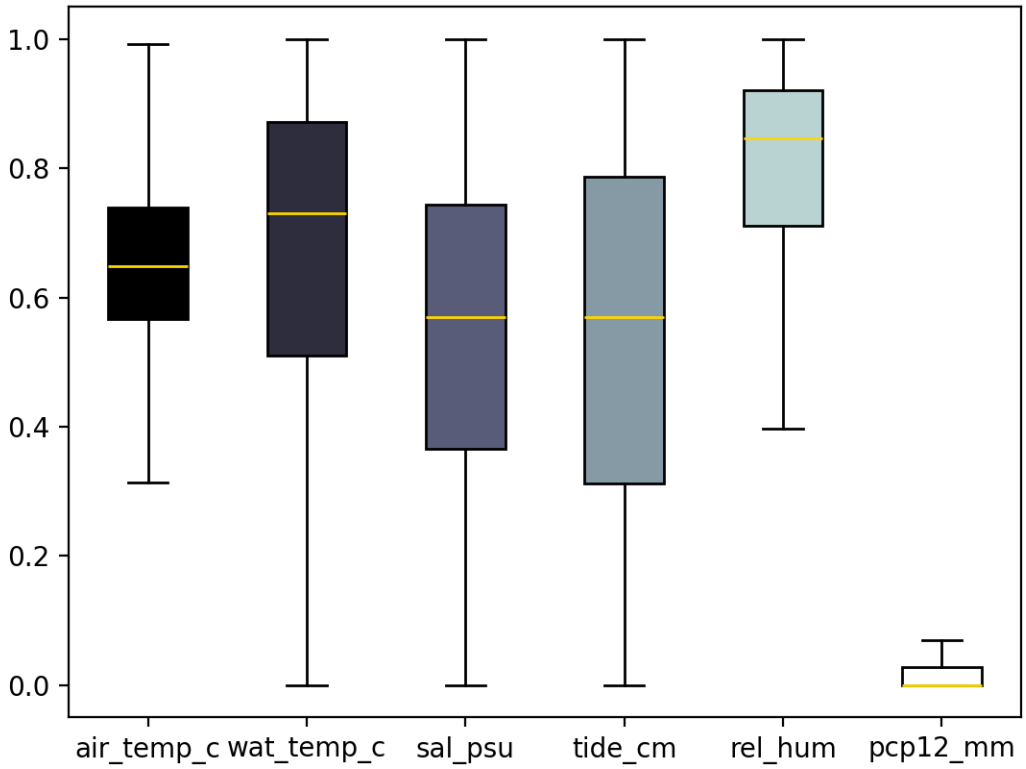
show notches

```
_ = boxplot(df[cols],
        fill_color="pink",
        notch=True,
        patch_artist=True,
        medianprops={"color": "black"})
```



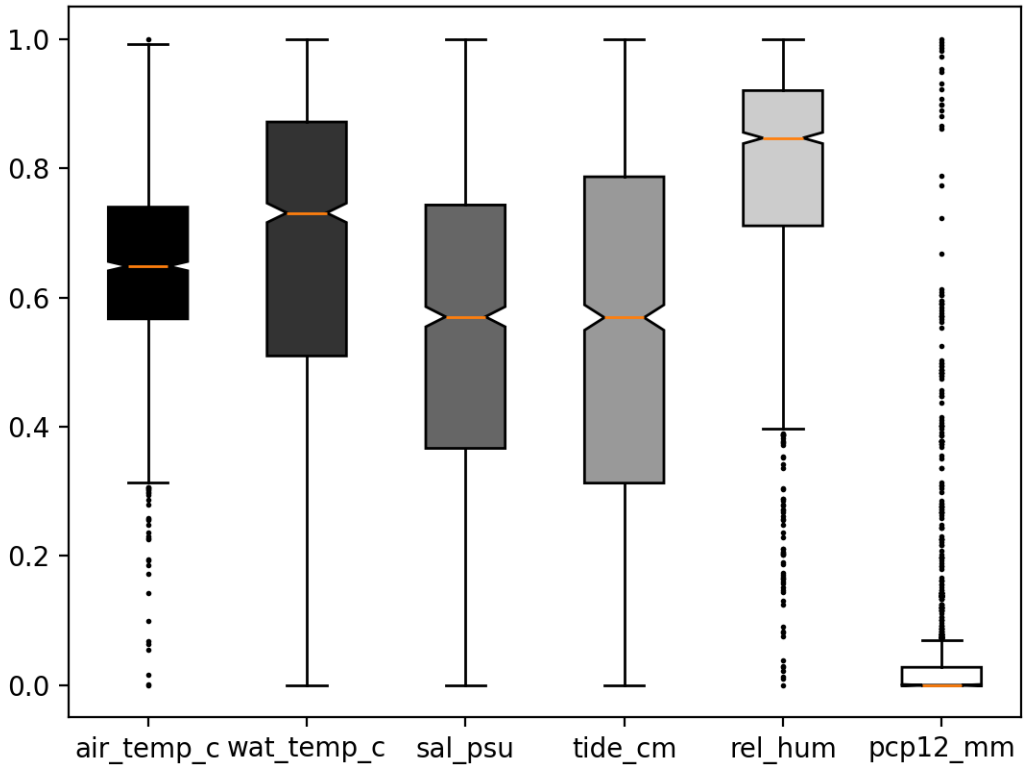
don't show outliers

```
_ = boxplot(df[cols], fill_color="bone", patch_artist=True, showfliers=False,
            medianprops={"color": "gold"})
```



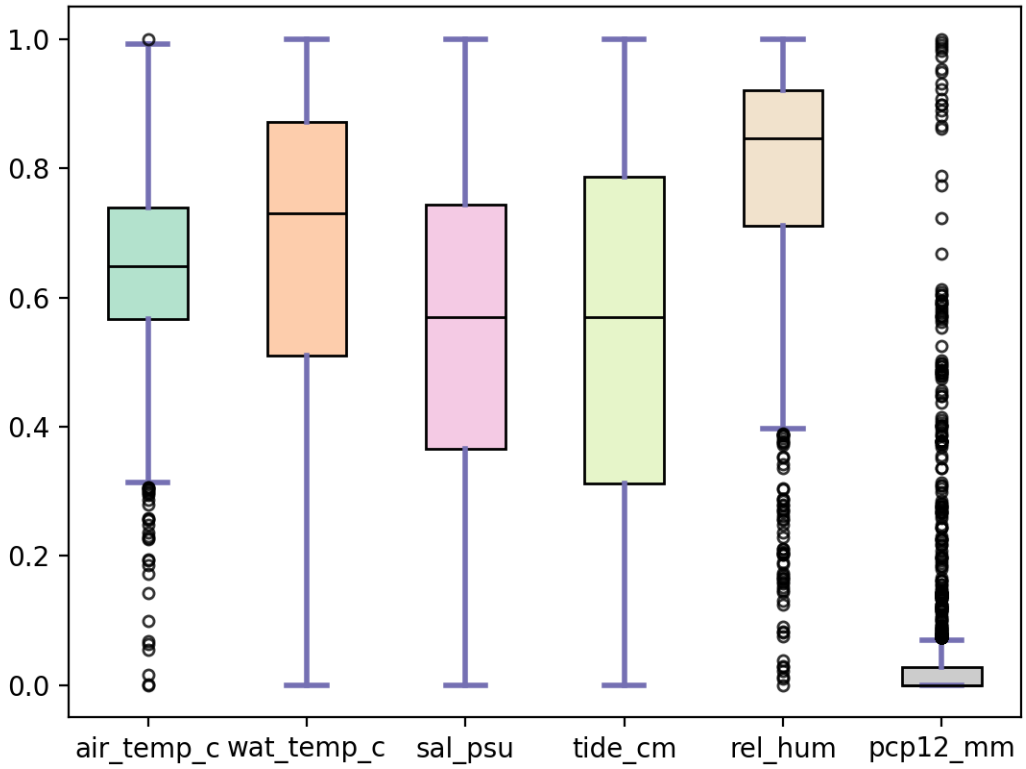
change circle size of fliers

```
_ = boxplot(df[cols], fill_color="gray", patch_artist=True,notch=True,
            flierprops={"ms": 1.0})
```



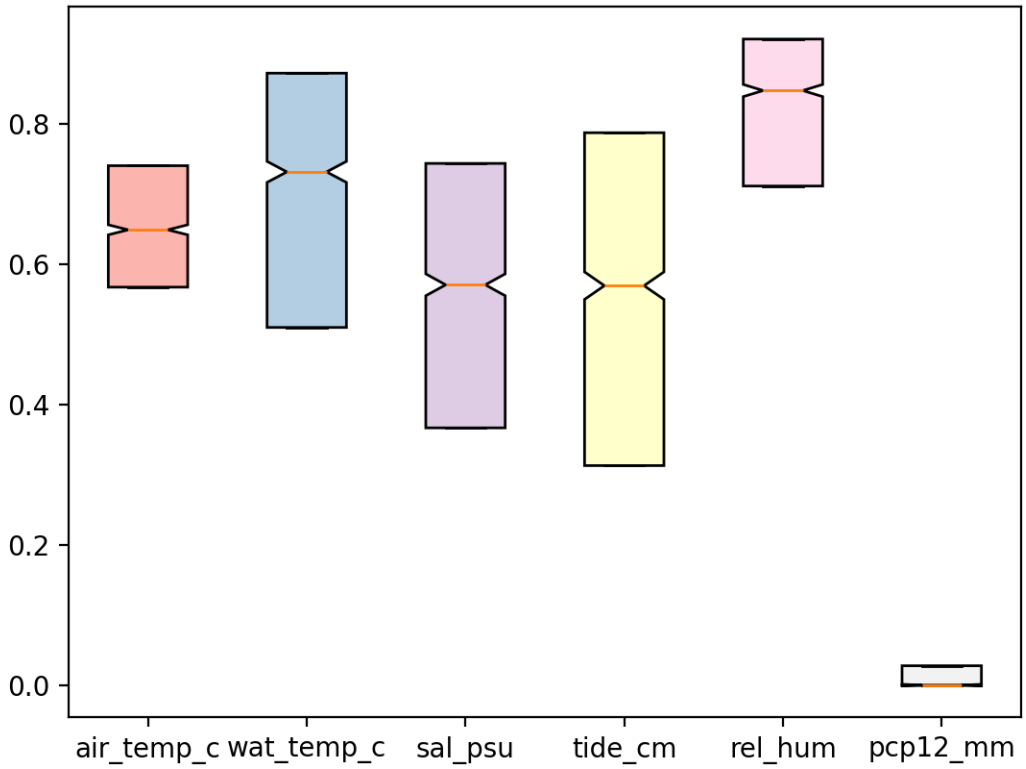
edit caps and whiskers properties

```
_ = boxplot(df[cols], fill_color="Pastel2", patch_artist=True,
            flierprops={"ms": 4.0,
                        "marker": 'o',
                        "color": 'thistle',
                        "alpha":0.8},
            medianprops={"color": "black"},
            capprops={'color':'#7570b3', "linewidth":2},
            whiskerprops={'color':'#7570b3', "linewidth":2})
```



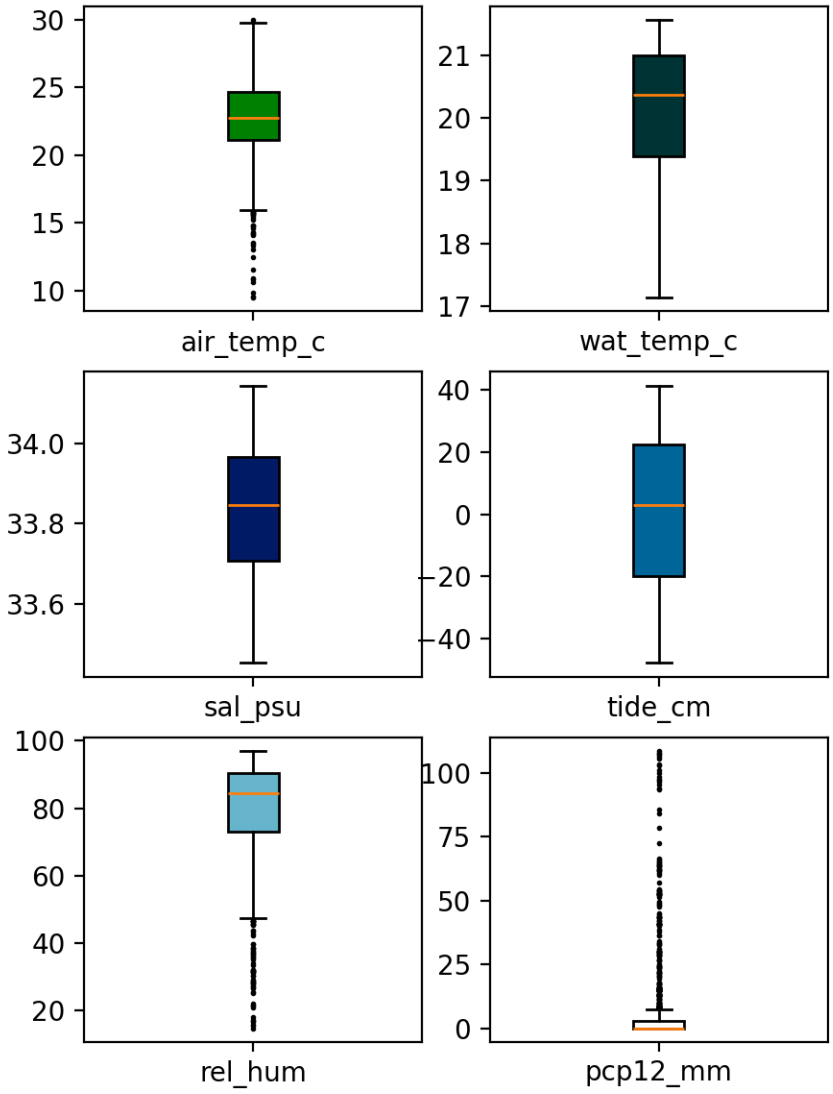
don't show whiskers

```
_ = boxplot(df[cols], fill_color="Pastel1",
            patch_artist=True, notch=True,
            showfliers=False, whis=0.0)
```



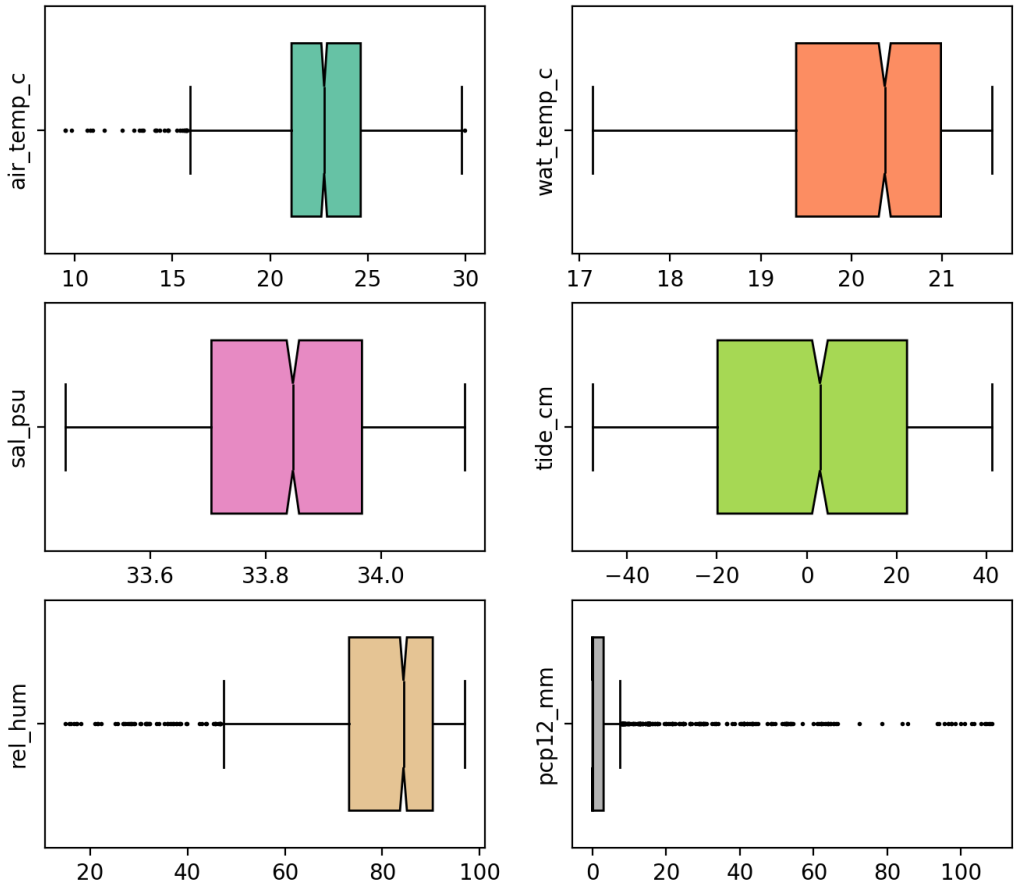
If we want to draw a separate boxplot on each axes, we can set the value of `share_axes` to False.

```
_ = boxplot(dataframe[cols], flierprops={"ms": 1.0},
            fill_color="ocean", patch_artist=True,
            share_axes=False, figsize=(5, 7))
```



To draw the horizontal boxplots, we can set `vert` to False.

```
_ = boxplot(dataframe[cols], flierprops={"ms": 1.0},
            fill_color="Set2", patch_artist=True, notch=True,
            medianprops={"color": "black"},
            share_axes=False, vert=False, widths=0.7,
            figsize=(8, 7)
            )
```



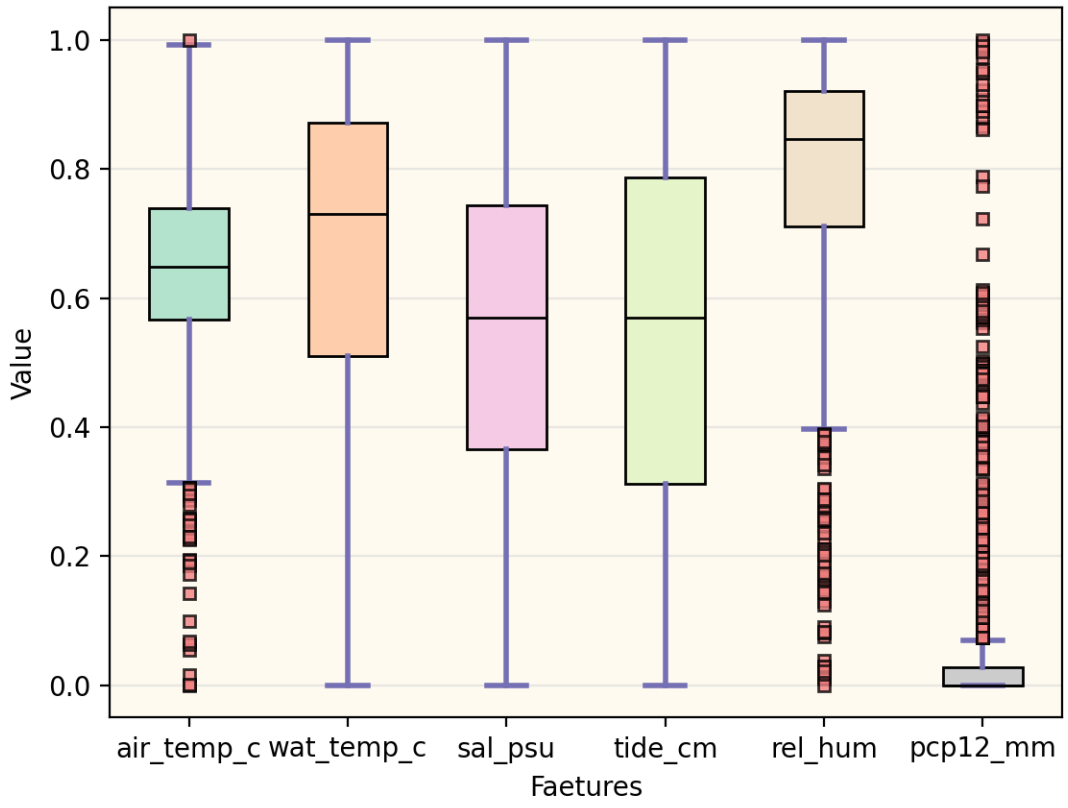
The boxplot function returns a tuple. The first argument is the matplotlib axes and second value is a dictionary consisting of output from `axes.boxplot`.

```
ax, _ = boxplot(df[cols], fill_color="Pastel2", patch_artist=True,
            flierprops={"ms": 4.0,
                        "marker": 's',
                        "markerfacecolor": 'lightcoral',
                        "alpha":0.8
                        },
            medianprops={"color": "black"},
            capprops={'color':'#7570b3', "linewidth":2},
            whiskerprops={'color':'#7570b3', "linewidth":2},
             show=False)

ax.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)
ax.set(
    axisbelow=True,  # Hide the grid behind plot objects
```

(continues on next page)

(continued from previous page)

```
    xlabel='Faetures',
    ylabel='Value',
)
ax.set_facecolor('floralwhite')
plt.show()
```



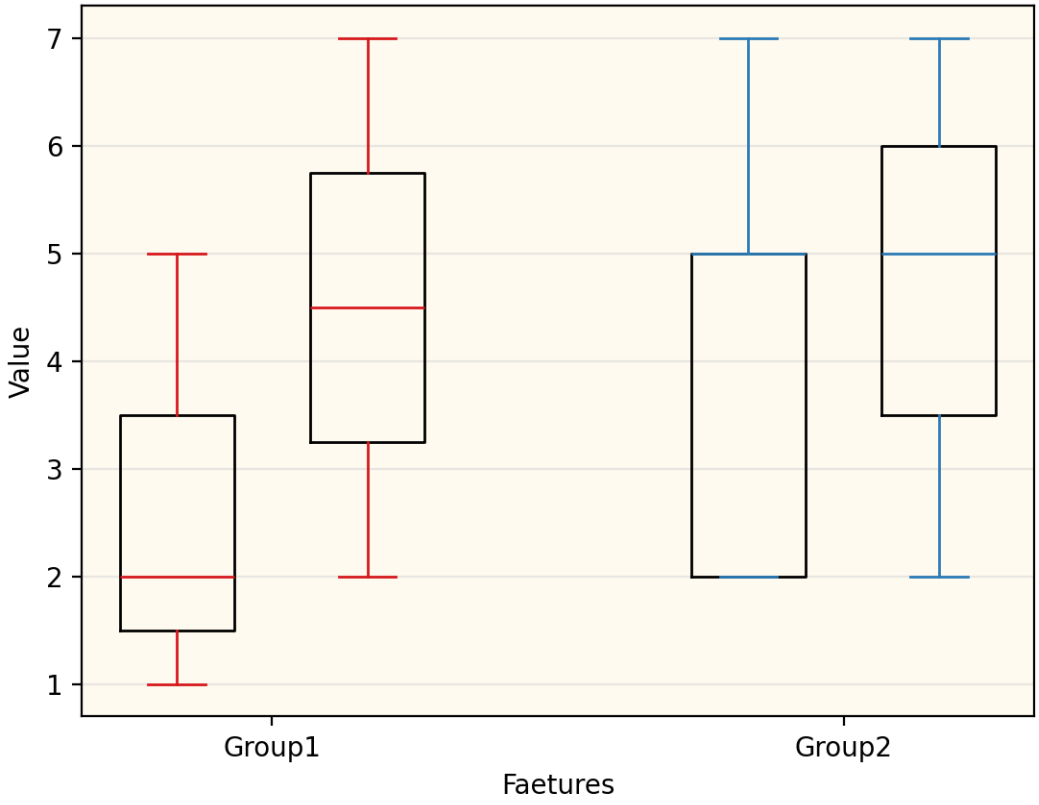
In order to make grouped boxplots, we can drawo two boxplots on same axes. We can specify the position of boxes on the axes using `positions` argument.

```
# Some fake data to plot
A= [[1, 2, 5,],  [7, 2]]
B = [[5, 7, 2, 2, 5], [7, 2, 5]]

ax, _ = boxplot(A, line_color='#D7191C', positions=[1, 2], sym='', widths = 0.6,
              show=False)

_, _ = boxplot(B, line_color="#2C7BB6", positions=[4, 5], sym='', widths = 0.6,
              show=False)

ax.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)
ax.set_xticks([1.5, 4.5])
ax.set_xticklabels(['Group1', 'Group2'])
```

(continues on next page)

(continued from previous page)

```
ax.set(
    axisbelow=True,  # Hide the grid behind plot objects
    xlabel='Faetures',
    ylabel='Value',
)

ax.set_facecolor('floralwhite')

plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/python-seekho/envs/latest/lib/python3.9/
↪site-packages/easy_mpl/_box.py:184: UserWarning:
xticks (4) and xticklabels (2) dont match
  warnings.warn(f"""
```

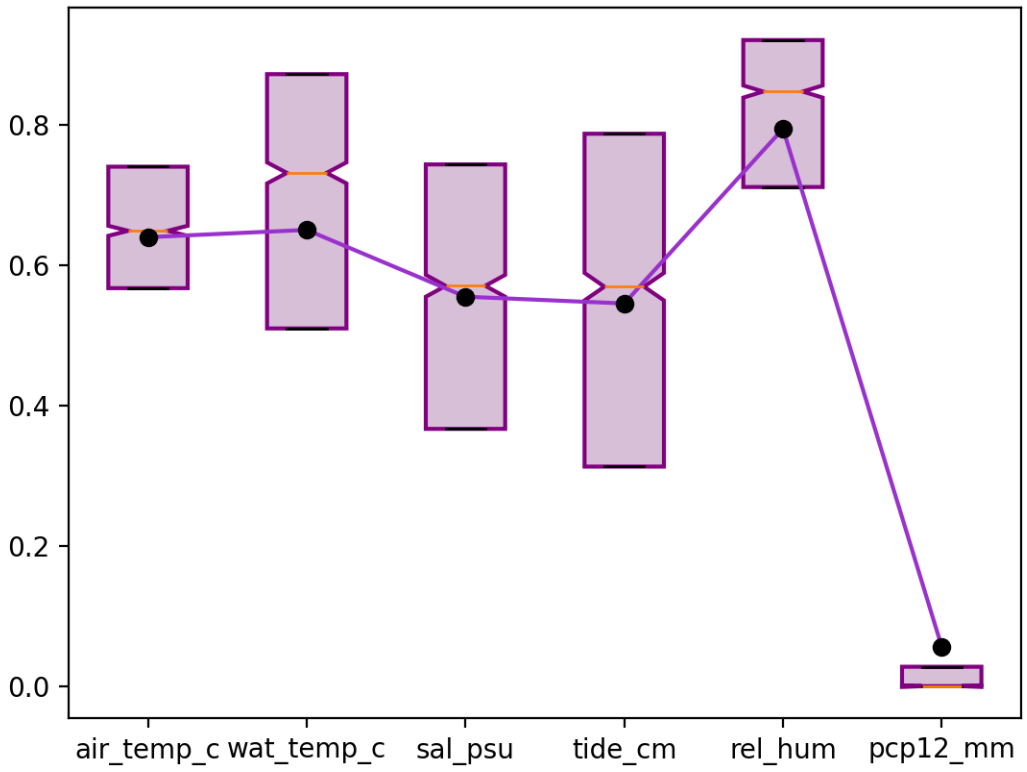
join mean of each box through a line

```
plt.close('all')
ax, _ = boxplot(df[cols], fill_color="thistle",
            patch_artist=True, notch=True,
            boxprops = {"linewidth":1.5,
                        "color":'purple'},
```

(continues on next page)

(continued from previous page)

```
            showmeans=True, meanprops={"markerfacecolor": "black",
                                       "markeredgecolor":'black',
                                        "marker":"o"},
            showfliers=False, whis=0.0,
            show=False)

plot(ax.get_xticks(), df[cols].mean().values,
     color="darkorchid", ax=ax)
```




```
<Axes: >
```

Total running time of the script: (0 minutes 5.025 seconds)

6.17 violin

This file shows the usage of `violin_plot()` function.

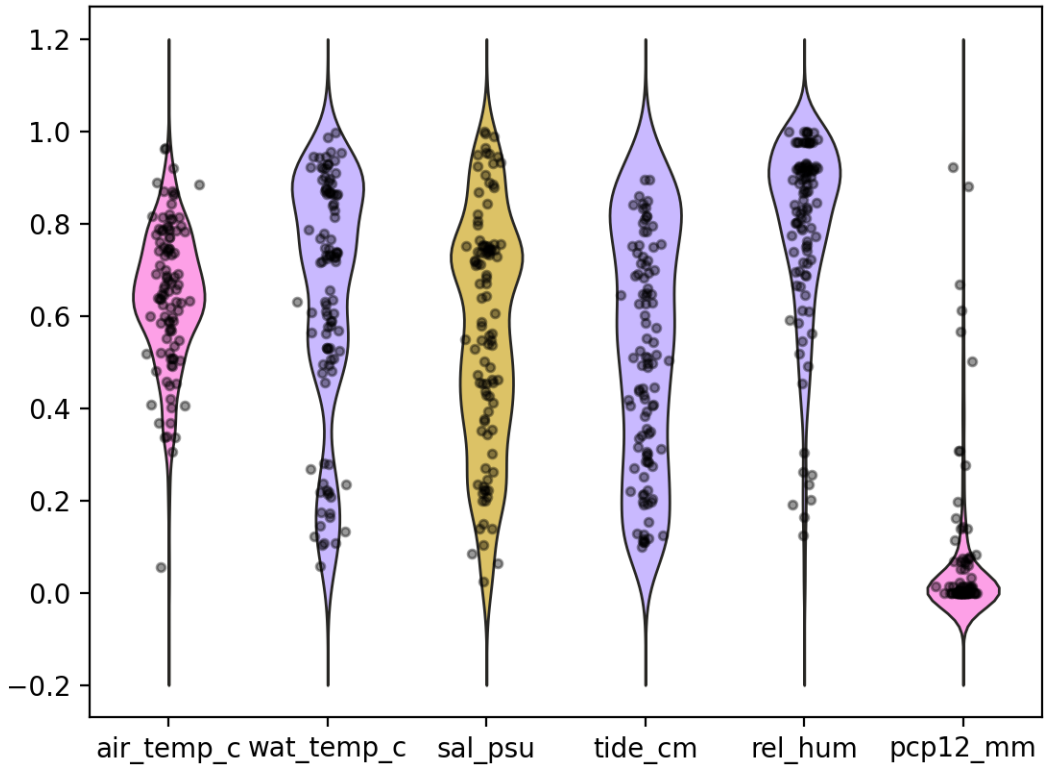
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from easy_mpl import violin_plot
from easy_mpl.utils import _rescale
from easy_mpl.utils import version_info

version_info()
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

```
f = "https://raw.githubusercontent.com/AtrCheema/AI4Water/master/ai4water/datasets/arg_
↪busan.csv"
df = pd.read_csv(f, index_col='index')
cols = ['air_temp_c', 'wat_temp_c', 'sal_psu', 'tide_cm', 'rel_hum', 'pcp12_mm']
for col in df.columns:
    df[col] = _rescale(df[col].values)
```

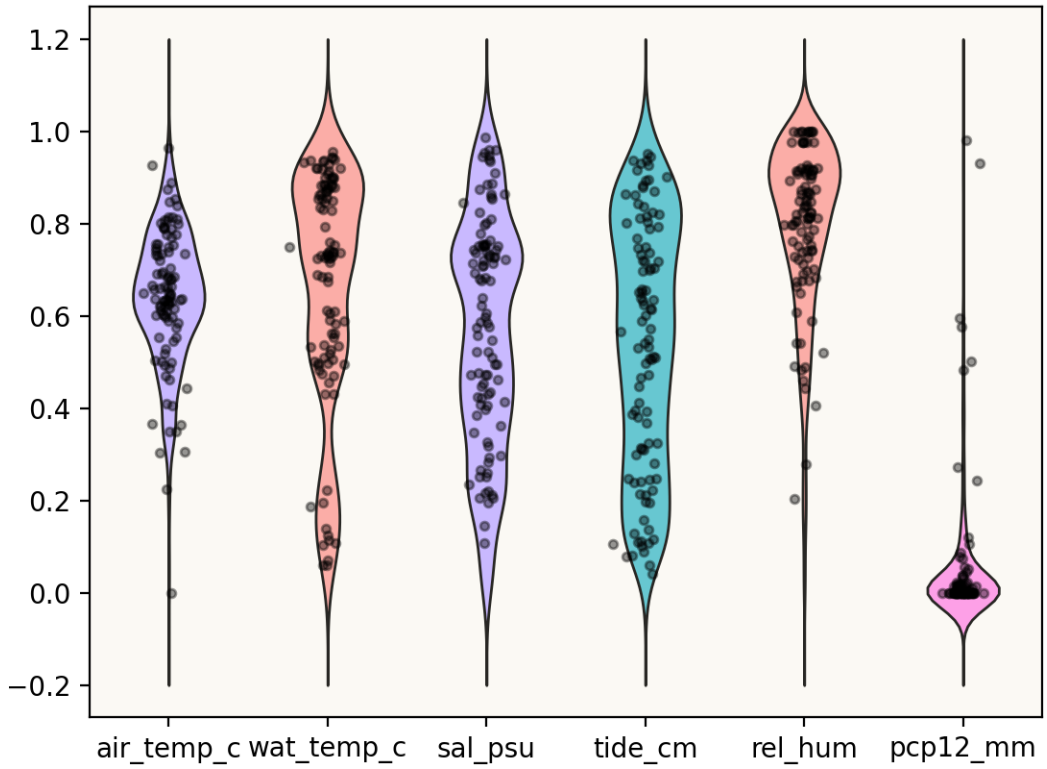
The basic violin plot can be drawn by passing a pandas DataFrame with one or more columns.

```
_ = violin_plot(df[cols])
```



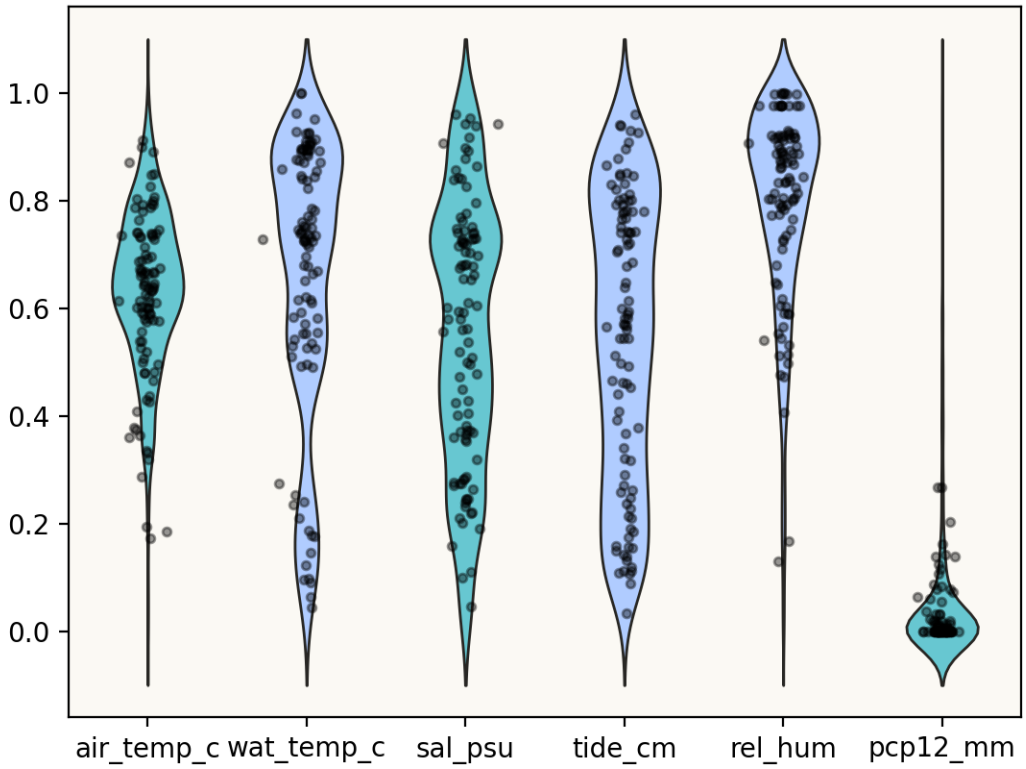
The function *:func: `violin_plot* always returns a matplotlib axes object. If we set `show` to False then, the returned axes can be used for further manipulation.

```
axes = violin_plot(df[cols], show=False)
axes.set_facecolor("#fbf9f4")
plt.show()
```



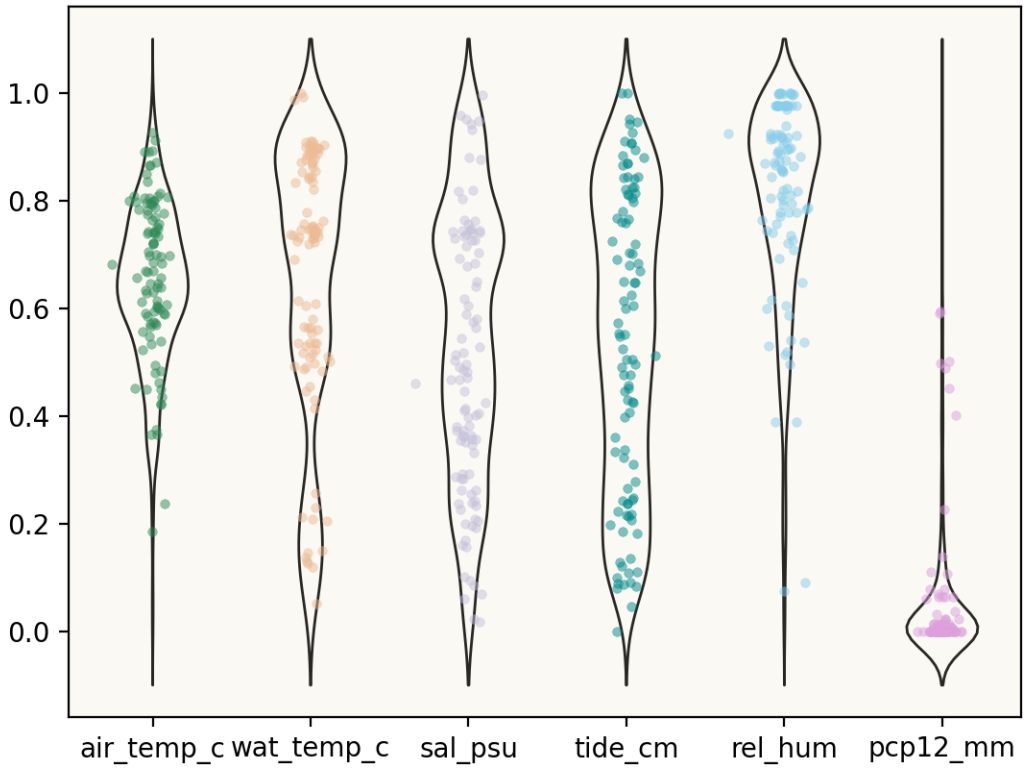
The value of `cut` determines the length of tails on both sides.

```
axes = violin_plot(df[cols], show=False, cut=0.1)
axes.set_facecolor("#fbf9f4")
plt.show()
```



We can specify colors for each of the violin plot separately. It can be any valid matplotlib colors i.e., name of color as string or its RGB vlaue.

```
axes = violin_plot(
    df[cols], show=False, cut=0.1, fill=False,
    scatter_kws={"s": 12, 'alpha': 0.5, 'edgecolors': None, 'linewidths': 0.2},
    datapoints_colors=['seagreen',
                       np.array([237, 187, 147]) / 255,
                      np.array([197, 194, 218]) / 255,
                      'darkcyan',
                      'skyblue',
                      "plum",
                       ]
                  )
axes.set_facecolor("#fbf9f4")
plt.show()
```



Total running time of the script: (0 minutes 1.468 seconds)

6.18 Adding Marginal plots

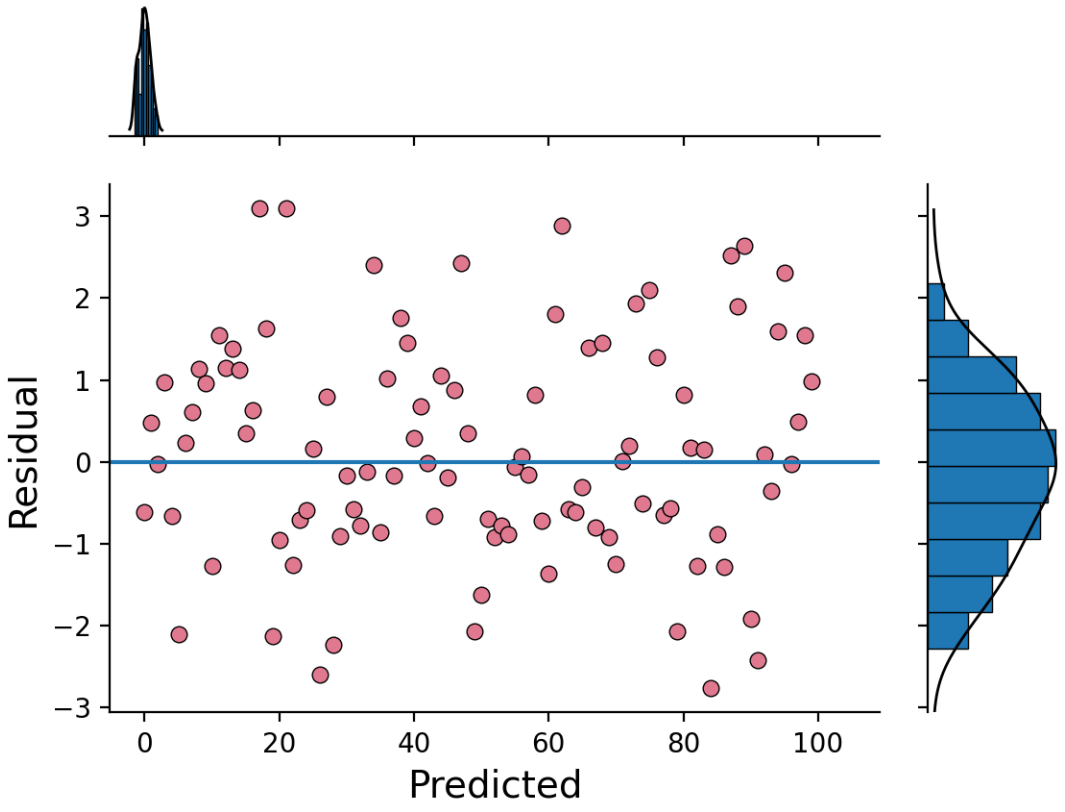
This lesson shows how to add marginal plots to an existing matplotlib axes

```
# sphinx_gallery_thumbnail_number = 2

import numpy as np
from easy_mpl import plot
from easy_mpl import regplot
import matplotlib.pyplot as plt
from easy_mpl.utils import version_info
from easy_mpl.utils import AddMarginalPlots

version_info()
```

```
{'easy_mpl': '0.21.4', 'matplotlib': '3.8.4', 'numpy': '1.26.4', 'pandas': '1.5.3',
↪'scipy': '1.13.1'}
```

We can add marginal plots to our main plot using `AddMarginalPlots` class. The marginal plots are used to show the distribution of x-axis data and y-axis data. The distribution of x-axis data is shown on top of main plot and the distribution of y-axis data is shown on right side of main plot.

```
x = np.random.normal(size=100)
y = np.random.normal(size=100)
e = x-y

ax = plot(
    e,
    'o',
    show=False,
    markerfacecolor=np.array([225, 121, 144])/256.0,
    markeredgecolor="black", markeredgewidth=0.5,
    ax_kws=dict(
        xlabel="Predicted",
        ylabel="Residual",
        xlabel_kws={"fontsize": 14},
        ylabel_kws={"fontsize": 14}),
     )

# draw horizontal line on y=0
ax.axhline(0.0)
AddMarginalPlots(ax)(x, y)
plt.show()
```

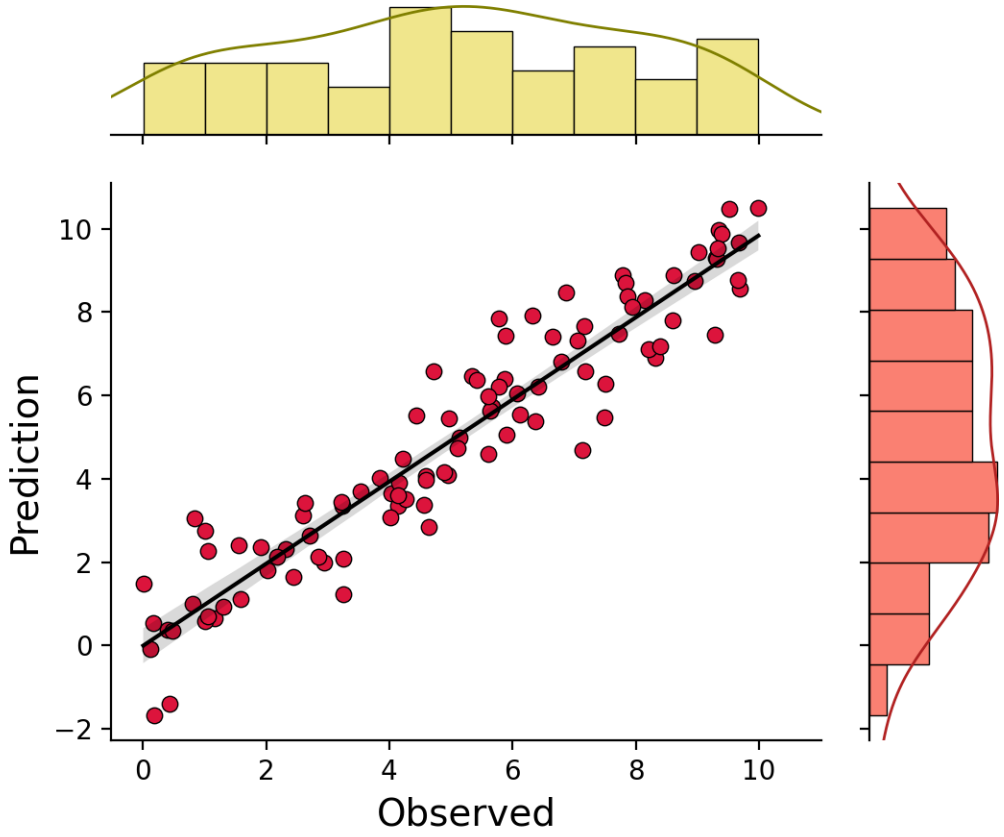



```
rng = np.random.default_rng(313)
```

(continues on next page)

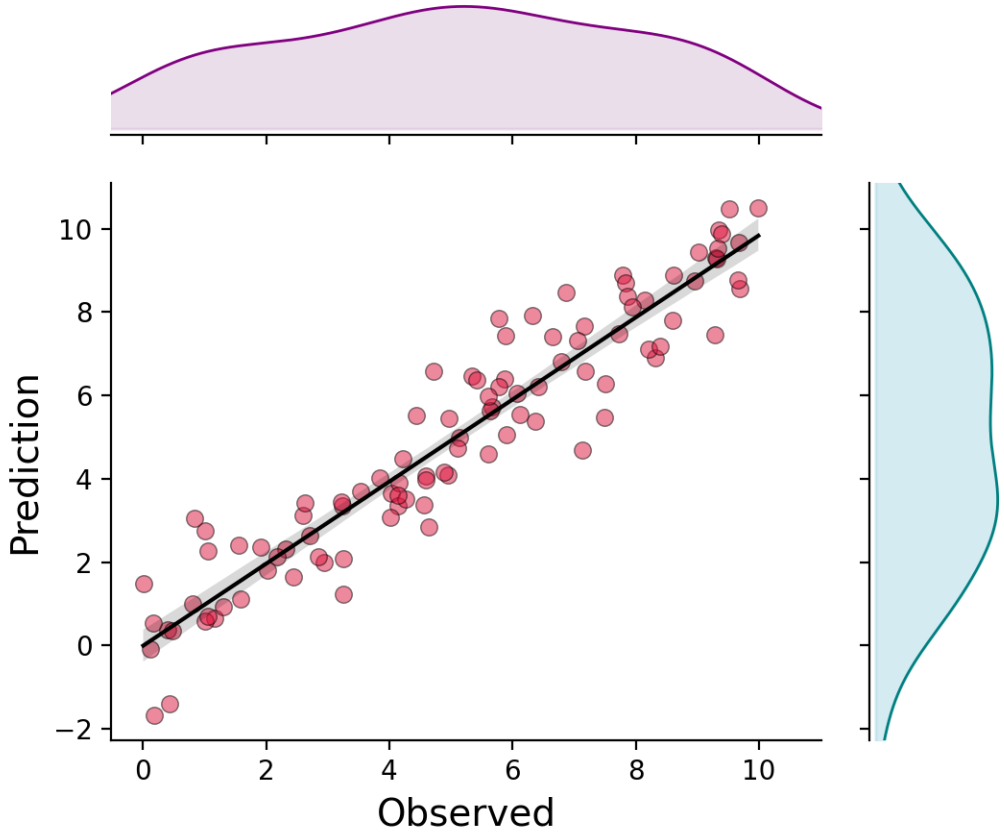
(continued from previous page)

```
x = rng.uniform(0, 10, size=100)
y = x + rng.normal(size=100)


# We can show distribution of x and y along the marginals
# This can be done by setting the ``marginals`` keyword to True

RIDGE_LINE_KWS = [{'color': 'olive', 'lw': 1.0}, {'color': 'firebrick', 'lw': 1.0}]
HIST_KWS = [{'color': 'khaki'}, {'color': 'salmon'}]

_ = regplot(x, y,
              marker_size = 35,
              marker_color='crimson',
              line_color='k',
              fill_color='k',
              scatter_kws={'edgecolors':'black', 'linewidth':0.5,
                           },
              marginals=True,
              marginal_ax_pad=0.25,
              marginal_ax_size=0.7,
              ridge_line_kws=RIDGE_LINE_KWS,
              hist=True,
              hist_kws=HIST_KWS)
```



Instead of drawing histograms, we can decide to fill the ridges drawn by kde lines on marginals.

```
fill_kws = [{'color': 'thistle'}, {'color': 'lightblue'}]
RIDGE_LINE_KWS1 = [{'color': 'purple', 'lw': 1.0}, {'color': 'teal', 'lw': 1.0}]

_ = regplot(x, y,
            marker_size = 40,
            marker_color='crimson',
            line_color='k',
            fill_color='k',
            scatter_kws={'edgecolors':'black', 'linewidth':0.5,
                         'alpha': 0.5},
            marginals=True,
            marginal_ax_pad=0.25,
            marginal_ax_size=0.7,
            ridge_line_kws=RIDGE_LINE_KWS1,
            hist=False,
            fill_kws=fill_kws)
```



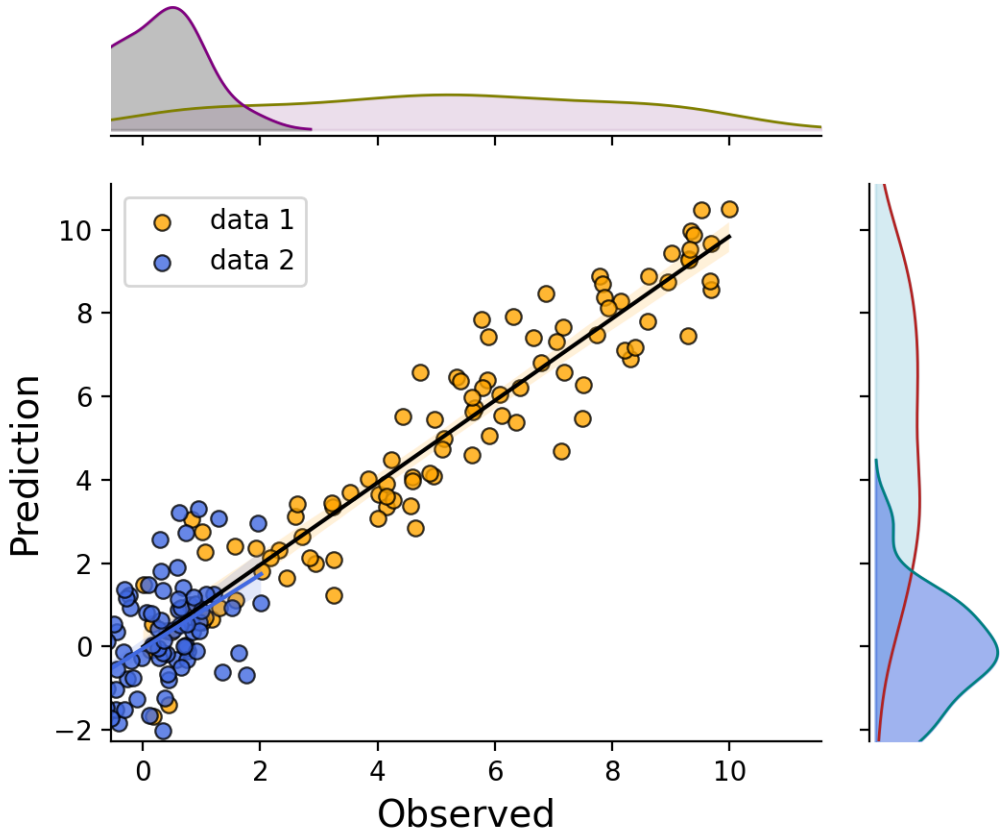
multiple regression lines with customized marker, line and fill style

```
cov = np.array(
    [[1.0, 0.9, 0.7],
     [0.9, 1.2, 0.8],
     [0.7, 0.8, 1.4]]
)
data = rng.multivariate_normal(np.zeros(3),
                               cov, size=100)

ax = regplot(x, y, line_color='k',
             marker_color='orange', marker_size=35, fill_color='orange',
             scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
             show=False, label="data 1")

axHistx, axHisty = AddMarginalPlots(
    ax, hist=False, fill_kws=fill_kws,
    ridge_line_kws=RIDGE_LINE_KWS
)(x, y)

fill_kws1 = [{'color': 'grey'}, {'color': 'royalblue'}]
_ = regplot(data[:, 0], data[:, 2], line_color='royalblue', ax=ax,
                marker_color='royalblue', marker_size=35, fill_color='royalblue',
```

(continues on next page)

(continued from previous page)

```
            scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
            show=False, label="data 2", ax_kws=dict(legend_kws=dict(loc=(0.1, 0.8))))

AddMarginalPlots(
    ax, hist=False,
    fill_kws=fill_kws1, ridge_line_kws=RIDGE_LINE_KWS1)(data[:, 0], data[:, 2], axHistx,␣
↪axHisty)

plt.show()
```



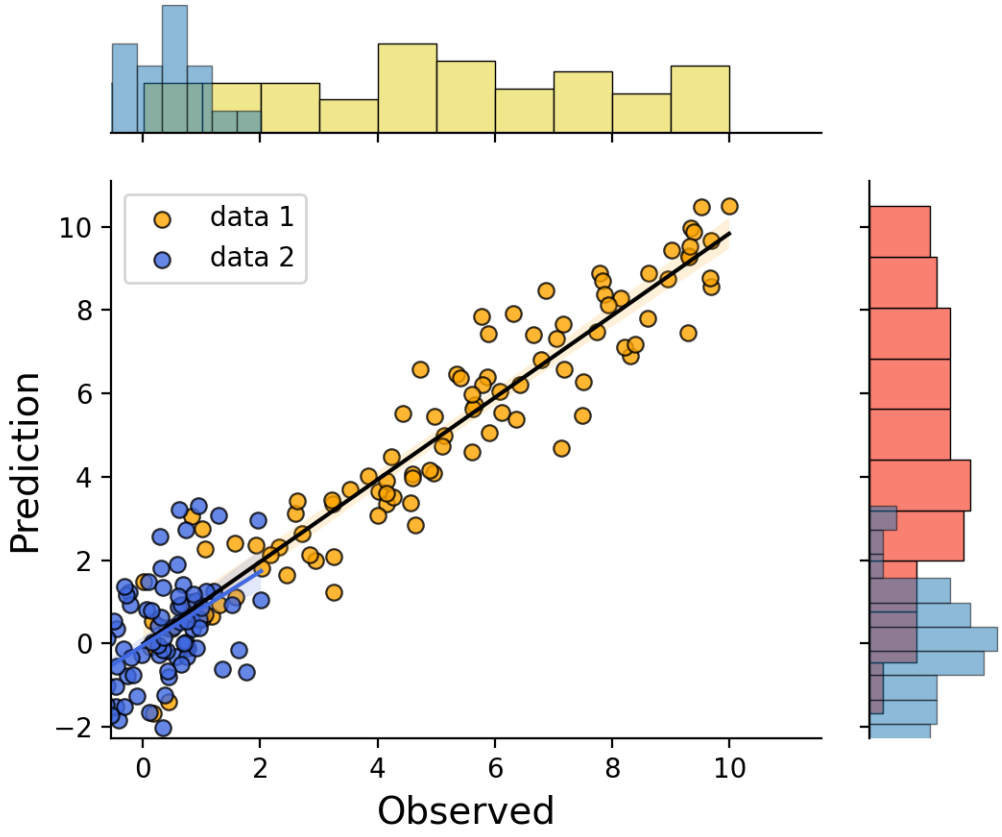
Showing distributions using histograms

```
ax = regplot(x, y, line_color='k',
             marker_color='orange', marker_size=35, fill_color='orange',
             scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
             show=False, label="data 1")

axHistx, axHisty = AddMarginalPlots(
    ax, ridge=False, hist_kws=HIST_KWS
)(x, y)
```

(continues on next page)

(continued from previous page)

```
fill_kws1 = [{'color': 'grey'}, {'color': 'royalblue'}]
_ = regplot(data[:, 0], data[:, 2], line_color='royalblue', ax=ax,
                marker_color='royalblue', marker_size=35, fill_color='royalblue',
             scatter_kws={'edgecolors':'black', 'linewidth':0.8, 'alpha': 0.8},
             show=False, label="data 2", ax_kws=dict(legend_kws=dict(loc=(0.1, 0.8))))

HIST_KWS1 = {'alpha': 0.5}
AddMarginalPlots(
    ax, ridge=False, hist_kws=HIST_KWS1
)(data[:, 0], data[:, 2], axHistx, axHisty)

plt.show()
```



Total running time of the script: (0 minutes 2.560 seconds)

2.7.7 7. Advanced

7.1 Regular expressions

Total running time of the script: (0 minutes 0.000 seconds)

7.2. Parallel Processing

This lesson demonstrates various ways of parallelizing the python code on cpu cores.

```
import time
from itertools import product
import concurrent.futures as cf
from multiprocessing import cpu_count, Pool

import numpy as np
import pandas as pd
import scipy.stats as ss
from numpy.random import default_rng
from joblib import Parallel, delayed
```

```
print(cpu_count())
```

```
2
```

Decide the number of cpu processes to use to paralllize the code. Parallelizing the code makes sense if we have at least 2 cores.

```
cpus = max(2, cpu_count()//2)
print(cpus)
```

```
2
```

We have two versions of *cramers_v* function. The first version takes a single argument *indices* as input. This function then uses the `data` from global scope to get two arrays (x and y) and calculates Cramers' v value.The second function, on the other hand uses the two arrays (x and y) as innput and calcuates Cramers` v for them.

```
def cramers_v(indices):
    i, j = indices
    x, y = data.loc[:, i], data.loc[:, j]
    return _cramers_v(x, y)


def cramers_v_data(x, y):
    return _cramers_v(x, y)


def _cramers_v(x, y):
    confusion_matrix = pd.crosstab(x, y)
    chi2 = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2 / n
```

(continues on next page)

(continued from previous page)

```
    r, k = confusion_matrix.shape
    phi2corr = max(0, phi2 - ((k - 1) * (r - 1)) / (n - 1))
    rcorr = r - ((r - 1) ** 2) / (n - 1)
    kcorr = k - ((k - 1) ** 2) / (n - 1)
    return np.sqrt(phi2corr / min((kcorr - 1), (rcorr - 1)))
```

Create data with categorical/string values. Increasing the number of columns will increase the number of for loops.

```
columns = 20  # total number fo for loops will be columns * columns
data = pd.DataFrame(np.random.randint(1, 20, (5000, columns)).astype(str))

print(data.shape)
```

```
(5000, 20)
```

Sequential Implementation

```
def without_pp():
    start = time.time()
    results = np.full((data.shape[1], data.shape[1]), fill_value=np.nan)
    for i in range(data.shape[1]):
        for j in  range(data.shape[1]):
            results[i,j] = cramers_v((i,j))
    print(round(time.time() - start, 2), 'seconds in sequential mode')
    return results
```

Parallel Implementations

We will compare 4 different ways of parallelizing the code.

ProcessPoolExecutor from concurrent

```
def with_ppe():

    start = time.time()
    with cf.ProcessPoolExecutor(max_workers=cpus) as executor:
        list_of_cols = list(product(range(data.shape[1]), range(data.shape[1])))
        results = executor.map(
            cramers_v,
            list_of_cols
        )
    print(round(time.time() - start, 2), 'seconds with ProcessPoolExecutor')

    return results
```

```
def with_ppe_data():
    rng = default_rng(313)
```

(continues on next page)

(continued from previous page)

```
    data_ = pd.DataFrame(rng.integers(1, 20, (5000, columns)).astype(str))

    start = time.time()
    with cf.ProcessPoolExecutor(max_workers=cpus) as executor:
        list_of_cols = list(product(range(data_.shape[1]), range(data_.shape[1])))
        list_of_x = [data_.loc[:, i] for (i, _) in list_of_cols]
        list_of_y = [data_.loc[:, j] for (_, j) in list_of_cols]
        results = executor.map(
            cramers_v_data,
            list_of_x,
            list_of_y
        )
    print(round(time.time() - start, 2), 'seconds with ProcessPoolExecutor (data)')
    return results
```

ThreadPoolExecutor from concurrent

```
def with_tpe():
    start = time.time()
    with cf.ThreadPoolExecutor(max_workers=cpus) as executor:
        list_of_cols = list(product(range(data.shape[1]), range(data.shape[1])))
        results = executor.map(
            cramers_v,
            list_of_cols
        )
    print(round(time.time() - start, 2), 'seconds with ThreadPoolExecutor')
    return results
```

```
def with_tpe_data():

    rng = default_rng(313)
    data_ = pd.DataFrame(rng.integers(1, 20, (5000, columns)).astype(str))

    start = time.time()
    with cf.ThreadPoolExecutor(max_workers=cpus) as executor:
        list_of_cols = list(product(range(data.shape[1]), range(data.shape[1])))
        list_of_x = [data_.loc[:, i] for (i,j) in list_of_cols]
        list_of_y = [data_.loc[:, j] for (i, j) in list_of_cols]

        results = executor.map(
            cramers_v_data,
            list_of_x,
            list_of_y
        )
    print(round(time.time() - start, 2), 'seconds with ThreadPoolExecutor (data)')
    return results
```

Pool from multiprocessing

```
def with_pool():

    start = time.time()
    with Pool(processes=cpus) as executor:
        list_of_cols = list(product(range(data.shape[1]), range(data.shape[1])))
        results = executor.map(
            cramers_v,
            list_of_cols,
        )
    print(round(time.time() - start, 2), 'seconds with Pool')

    return results
```

The following function uses Pool but with two changes. First, the function cramers_v_data receives the actual arrays (of data) instead of indices. In this the function which we want to parallelize, does not uses any global variable/data. The second difference is that our function now receives two inputs instead of a single input.

```
def with_pool_data():

    rng = default_rng(313)
    data_ = pd.DataFrame(rng.integers(1, 20, (5000, columns)).astype(str))

    start = time.time()
    with Pool(processes=cpus) as executor:
        list_of_cols = list(product(range(data_.shape[1]), range(data_.shape[1])))
        list_of_xy = [(data_.loc[:, i], data_.loc[:, j]) for (i,j) in list_of_cols]
        results = executor.starmap(
            cramers_v_data,
            list_of_xy,
        )
    print(round(time.time() - start, 2), 'seconds with Pool (data)')
    return results
```

Using joblib

```
def with_joblib(backend="loky", verbose=0, batch_size="auto"):
    start = time.time()
    list_of_cols = list(product(range(data.shape[1]), range(data.shape[1])))
    results = Parallel(
        n_jobs=cpus, backend=backend, verbose=verbose, batch_size=batch_size)(
        delayed(cramers_v)(val) for val in list_of_cols)

    print(round(time.time() - start, 2), f'seconds with joblib backend {backend}')
    return results
```

```
def with_joblib_and_data(backend="loky", verbose=0, batch_size="auto"):

    rng = default_rng(313)
    data_ = pd.DataFrame(rng.integers(1, 20, (5000, columns)).astype(str))
```

(continues on next page)

(continued from previous page)

```
    start = time.time()
    list_of_cols = list(product(range(data_.shape[1]), range(data_.shape[1])))
    results = Parallel(
        n_jobs=cpus, backend=backend, verbose=verbose, batch_size=batch_size)(
        delayed(cramers_v_data)(
            data_.loc[:, i], data_.loc[j]) for (i,j) in list_of_cols)

    print(round(time.time() - start, 2), f"seconds with joblib (data) backend {backend}")
    return results
```

```
if __name__ == "__main__":

    results_nopp = without_pp()

    results_ppe = with_ppe()
    results_ppe = np.array(list(results_ppe)).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_ppe))

    res_ppe_data = with_ppe_data()
    res_ppe_data = np.array(list(res_ppe_data)).reshape(results_nopp.shape)

    results_tpe = with_tpe()
    results_tpe = np.array(list(results_tpe)).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_tpe))

    res_tpe_data = with_tpe_data()
    res_tpe_data = np.array(list(res_tpe_data)).reshape(results_nopp.shape)
    print(np.allclose(res_ppe_data, res_tpe_data))

    results_pool = with_pool()
    results_pool = np.array(list(results_pool)).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_pool))

    res_pool_data = with_pool_data()
    res_pool_data = np.array(list(res_pool_data)).reshape(results_nopp.shape)
    print(np.allclose(res_ppe_data, res_pool_data))

    results_joblib = with_joblib()
    results_joblib = np.array(results_joblib).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_joblib))

    results_joblib = with_joblib(backend="multiprocessing")
    results_joblib = np.array(results_joblib).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_joblib))

    results_joblib = with_joblib(backend="threading")
    results_joblib = np.array(results_joblib).reshape(results_nopp.shape)
    print(np.allclose(results_nopp, results_joblib))

    res_joblib_data = with_joblib_and_data(backend="loky")
    res_joblib_data = np.array(res_joblib_data).reshape(results_nopp.shape)
```

(continues on next page)

(continued from previous page)

```
    print(np.allclose(res_ppe_data, res_joblib_data))

    res_joblib_data = with_joblib_and_data(backend="threading")
    res_joblib_data = np.array(res_joblib_data).reshape(results_nopp.shape)
    print(np.allclose(res_ppe_data, res_joblib_data))

# %%
```

```
5.38 seconds in sequential mode
5.07 seconds with ProcessPoolExecutor
True
5.5 seconds with ProcessPoolExecutor (data)
5.99 seconds with ThreadPoolExecutor
True
5.98 seconds with ThreadPoolExecutor (data)
True
4.98 seconds with Pool
True
5.08 seconds with Pool (data)
True
8.53 seconds with joblib backend loky
True
5.17 seconds with joblib backend multiprocessing
True
6.11 seconds with joblib backend threading
True
2.92 seconds with joblib (data) backend loky
False
3.84 seconds with joblib (data) backend threading
False
```

Total running time of the script: (1 minutes 4.750 seconds)

7.3 cython

Total running time of the script: (0 minutes 0.000 seconds)

7.4 interfacing with C

This lesson describes how to call functions/modules written in C into python and

```
import numpy as np

x= np.array([1, 2, 4,  5,  10, 20, 21, 22, 23, 24])
y= np.array([4, 6, 12, 15, 34, 68, 71, 72, 73, 74])

# perform linear regression
```

Total running time of the script: (0 minutes 0.001 seconds)

7.5 interfacing with C++

This lesson describes how to call functions/modules written in C++ into python and

```
import time
import ctypes
import subprocess
from ctypes import c_char_p, c_uint, c_size_t, POINTER, Structure

import numpy as np
import pandas as pd
```

```
start = time.time()
for i in range(100):
    pd.read_csv('daily_q.csv', index_col=0)

print(f"with pandas: {time.time() - start:.2f} seconds")
```

```
with pandas: 3.36 seconds
```

```
# Specify the C++ source file and the output executable name
cpp_source_file = 'read_csv_gpt.cpp'
output_library = 'read_csv_gpt.so'

# Compile the C++ code as a library
compile_command = f'g++ -shared -o {output_library}  -fPIC {cpp_source_file}'
process = subprocess.run(compile_command, shell=True, text=True, capture_output=True)

if process.returncode != 0:
    print(process.stderr)
else:
    print('Compilation successful')

# Define the CSVRow structure
class CSVRow(Structure):

    _fields_ = [("date", ctypes.c_char * 20),
                ("floatValue", ctypes.c_float)]

# Load the shared library
csv_reader = ctypes.CDLL(f'./{output_library}')  # Use the correct path for your .so/.
↪dll file

# Define the argument and return types of the functions
csv_reader.processCSVFile.argtypes = [c_char_p, POINTER(c_uint), POINTER(c_size_t)]
csv_reader.processCSVFile.restype = POINTER(CSVRow)

csv_reader.freeCSVRows.argtypes = [POINTER(CSVRow)]
csv_reader.freeCSVRows.restype = None

def read_csv(file_path):
    # Call the processCSVFile function
    station = c_uint(0)
```

(continues on next page)

(continued from previous page)

```
    size = c_size_t(0)
    rows = csv_reader.processCSVFile(file_path, ctypes.byref(station), ctypes.
↪byref(size))

    # Access the returned data
    data = [None] * size.value
    index = [None] * size.value
    for i in range(size.value):
        index[i] = rows[i].date.decode('utf-8')
        data[i] = rows[i].floatValue

    df = pd.DataFrame(data, columns=[str(station.value)], index=index)
    df.replace(-9999.0, np.nan, inplace=True)

    # Free the allocated memory
    csv_reader.freeCSVRows(rows)

    return df

df = read_csv(b'daily_q.csv')

start = time.time()
for i in range(100):
    read_csv(b'daily_q.csv')

print(f"With C++: {time.time() - start:.2f} seconds")
```

```
Compilation successful
With C++: 7.45 seconds
```

```
# Load the shared library
csv_reader = ctypes.CDLL(f'./{output_library}')   # Use the correct path for your .so/.
↪dll file

# # Define the argument and return types of the functions
csv_reader.processCSVFileFast.argtypes = [ctypes.c_char_p, ctypes.POINTER(ctypes.c_uint),
↪ ctypes.POINTER(ctypes.c_size_t), ctypes.POINTER(ctypes.c_size_t)]
csv_reader.processCSVFileFast.restype = ctypes.c_void_p

csv_reader.freeCSVRowsFast.argtypes = [ctypes.c_void_p]
csv_reader.freeCSVRowsFast.restype = None
```

```
def read_csvFast(file_path):
    # Call the processCSVFileFast function
    total = ctypes.c_uint(0)
    size = ctypes.c_size_t(0)
    row_size = ctypes.c_size_t(0)
    ptr = csv_reader.processCSVFileFast(file_path, ctypes.byref(total), ctypes.
↪byref(size), ctypes.byref(row_size))

    # # Check if the pointer is NULL
```

(continues on next page)

(continued from previous page)

```
    # if not ptr:
    #     raise MemoryError("Failed to allocate memory for CSV data")

    # Convert the pointer to a numpy array
    buffer = (ctypes.c_char * (size.value * row_size.value)).from_address(ptr)
    data = np.frombuffer(buffer, dtype=[('date', 'S20'), (str(total.value), 'f4')])

    # Create a pandas DataFrame from the numpy array
    df = pd.DataFrame(data)

    # Convert the 'date' column to string
    df.index = df['date'].str.decode('utf-8')

    # Free the allocated memory
    csv_reader.freeCSVRowsFast(ptr)
    return df
```

```
start = time.time()
for i in range(100):
    df = read_csvFast(b'daily_q.csv')

print(f"With C++: {time.time() - start:.2f} seconds")
```

```
With C++: 5.48 seconds
```

```
def read_csvFast1(file_path):
    # Call the processCSVFileFast function
    total = ctypes.c_uint(0)
    size = ctypes.c_size_t(0)
    row_size = ctypes.c_size_t(0)
    ptr = csv_reader.processCSVFileFast(file_path, ctypes.byref(total), ctypes.
↪byref(size), ctypes.byref(row_size))

    # Convert the pointer to a numpy array
    buffer = (ctypes.c_char * (size.value * row_size.value)).from_address(ptr)
    data = np.frombuffer(buffer, dtype=[('date', 'S20'), (str(total.value), 'f4')])

    # Convert the 'date' column to string
    #index = df['date'].str.decode('utf-8')

    # Free the allocated memory
    csv_reader.freeCSVRowsFast(ptr)
    return data
```

```
start = time.time()
for i in range(100):
    df = read_csvFast1(b'daily_q.csv')

print(f"With C++ in numpy: {time.time() - start:.2f} seconds")
```

```
With C++ in numpy: 2.38 seconds
```

Total running time of the script: (0 minutes 19.278 seconds)

7.6 interfacing with Fortran

This lesson describes how to call functions/modules written in Fortran into python and

Total running time of the script: (0 minutes 0.000 seconds)

7.7 interfacing with matlab

This lesson describes how to call functions/modules written in matlab into python and

Total running time of the script: (0 minutes 0.000 seconds)

7.8 testing your code

This lesson describes unit testing in python.

If you are writing a code which you think will not be used by you or anyone else in the future, then you can skip this lesson. However, if you intend to write a code, which will used by you and others in future then you must write tests for your code, so that your code is reliable. The purpose of writing tests is **to make sure that the code works and will keep on working the way it should work**.

Consider the following function which adds two numbers

```
def add(a,b):
    return a+b
```

Now if we want to write a test for this function, it will look something like below

```
summation = add(2,2)
assert summation==4
```

Above we are testing the `add` function. But we have tested it only with integers as input. We should probably check it with float or other types as well.

```
assert add(2.0, 2.0)==4.0
```

Total running time of the script: (0 minutes 0.001 seconds)

7.9 Speeding up with numba

```
import os
import time
from typing import Tuple
from multiprocessing import cpu_count

import sys
import numpy as np
from numba import numba
```

(continues on next page)

(continued from previous page)

```
from numba import jit, float32, int32, prange
from numba.types import Tuple as nbTuple
from numba import get_num_threads, set_num_threads
```

```
print("Python version: ", sys.version)
print("numpy version: ", np.__version__)
print("numba version: ", numba.__version__)
```

```
Python version:  3.9.19 (main, Jun 18 2024, 09:35:09)
[GCC 11.4.0]
numpy version:  1.26.4
numba version:  0.60.0
```

```
cpus = max(2, cpu_count()//2)
# os.environ['NUMBA_NUM_THREADS'] = str(cpus)  # Change it to the number of CPU cores you␣
↪want to use
# set_num_threads(cpus)
print("Number of threads:", get_num_threads())
```

```
Number of threads: 2
```

```
def prepare_data(
        data: np.ndarray,
        lookback: int,
        num_inputs: int = None,
        num_outputs: int = None,
        input_steps: int = 1,
        forecast_step: int = 0,
        forecast_len: int = 1,
) -> Tuple[np.ndarray, np.ndarray]:
    """

    """
    assert isinstance(data, np.ndarray)

    if num_inputs is None and num_outputs is None:
        raise ValueError("""
Either of num_inputs or num_outputs must be provided.
""")

    features = data.shape[1]
    if num_outputs is None:
        num_outputs = features - num_inputs

    if num_inputs is None:
        num_inputs = features - num_outputs

    assert num_inputs + num_outputs == features, f"""
num_inputs {num_inputs} + num_outputs {num_outputs} != total features {features}"""

    if len(data) <= 1:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Can not create batches from data with shape {data.shape}")

    time_steps = lookback

    examples = len(data)

    x = []
    y = []

    for i in range(examples - lookback * input_steps + 1 - forecast_step - forecast_len␣
↪+ 1):
        stx, enx = i, i + lookback * input_steps
        x_example = data[stx:enx:input_steps, 0:features - num_outputs]

        sty = (i + time_steps * input_steps) + forecast_step - input_steps
        eny = sty + forecast_len
        target = data[sty:eny, features - num_outputs:]

        x.append(np.array(x_example))
        y.append(np.array(target))

    if len(x)<1:
        raise ValueError(f"""
        no examples generated from data of shape {data.shape} with lookback
        {lookback} input_steps {input_steps} forecast_step {forecast_step} forecast_len
↪{forecast_len}
""")
    x = np.stack(x)
    # transpose because we want labels to be of shape (examples, outs, forecast_len)
    y = np.array([np.array(i, dtype=np.float32).T for i in y], dtype=np.float32)


    return x, y
```

```
@jit(nopython=True)
def prepare_data1(
        data: np.ndarray,
        lookback: int,
        num_inputs: int = None,
        num_outputs: int = None,
        input_steps: int = 1,
        forecast_step: int = 0,
        forecast_len: int = 1,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Prepare input and output data for time series forecasting.
    """
    if num_inputs is None and num_outputs is None:
        raise ValueError("Either of num_inputs or num_outputs must be provided.")

    features = data.shape[1]
    if num_outputs is None:
```

(continues on next page)

(continued from previous page)

```
        num_outputs = features - num_inputs

    if num_inputs is None:
        num_inputs = features - num_outputs

    if num_inputs + num_outputs != features:
        raise ValueError(f"num_inputs {num_inputs} + num_outputs {num_outputs} != total␣
↪features {features}")

    if len(data) <= 1:
        raise ValueError(f"Cannot create batches from data with shape {data.shape}")

    examples = len(data)
    max_examples = examples - lookback * input_steps + 1 - forecast_step - forecast_len␣
↪+ 1
    if max_examples <= 0:
        raise ValueError(f"No examples generated from data of shape {data.shape} with␣
↪lookback {lookback}, input_steps {input_steps}, forecast_step {forecast_step},␣
↪forecast_len {forecast_len}")

    x = np.empty((max_examples, lookback, num_inputs), dtype=data.dtype)
    y = np.empty((max_examples, forecast_len, num_outputs), dtype=data.dtype)

    for i in range(max_examples):
        x_start_idx = i
        x_end_idx = x_start_idx + lookback * input_steps
        y_start_idx = x_end_idx + forecast_step - input_steps
        y_end_idx = y_start_idx + forecast_len

        x[i] = data[x_start_idx:x_end_idx:input_steps, :num_inputs]
        y[i] = data[y_start_idx:y_end_idx, -num_outputs:]

    y = np.transpose(y, (0, 2, 1)).astype(np.float32)

    return x, y
```

```
@jit(nbTuple((float32[:, :, :], float32[:, :, :]))(float32[:, :], int32, int32, int32,␣
↪int32, int32, int32), nopython=True)
def prepare_data2(
        data: np.ndarray,
        lookback: int,
        num_inputs: int,
        num_outputs: int,
        input_steps: int,
        forecast_step: int,
        forecast_len: int,
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Prepare input and output data for time series forecasting.
    """
    features = data.shape[1]
```

(continues on next page)

(continued from previous page)

```
    if num_inputs + num_outputs != features:
        raise ValueError(f"num_inputs {num_inputs} + num_outputs {num_outputs} != total␣
↪features {features}")

    if len(data) <= 1:
        raise ValueError(f"Cannot create batches from data with shape {data.shape}")

    examples = len(data)
    max_examples = examples - lookback * input_steps + 1 - forecast_step - forecast_len␣
↪+ 1
    if max_examples <= 0:
        raise ValueError(f"No examples generated from data of shape {data.shape} with␣
↪lookback {lookback}, input_steps {input_steps}, forecast_step {forecast_step},␣
↪forecast_len {forecast_len}")

    x = np.empty((max_examples, lookback, num_inputs), dtype=np.float32)
    y = np.empty((max_examples, forecast_len, num_outputs), dtype=np.float32)

    for i in range(max_examples):
        x_start_idx = i
        x_end_idx = x_start_idx + lookback * input_steps
        y_start_idx = x_end_idx + forecast_step - input_steps
        y_end_idx = y_start_idx + forecast_len

        x[i] = data[x_start_idx:x_end_idx:input_steps, :num_inputs]
        y[i] = data[y_start_idx:y_end_idx, -num_outputs:]

    y = np.transpose(y, (0, 2, 1)).astype(np.float32)

    return x, y
```

```
@jit(nbTuple((float32[:, :, :], float32[:, :, :]))(float32[:, :], int32, int32, int32,␣
↪int32, int32, int32), nopython=True, parallel=True)
def prepare_data3(
        data: np.ndarray,
        lookback: int,
        num_inputs: int,
        num_outputs: int,
        input_steps: int,
        forecast_step: int,
        forecast_len: int,
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Prepare input and output data for time series forecasting.
    """
    features = data.shape[1]

    if num_inputs + num_outputs != features:
        raise ValueError(f"num_inputs {num_inputs} + num_outputs {num_outputs} != total␣
↪features {features}")

    if len(data) <= 1:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Cannot create batches from data with shape {data.shape}")

    examples = len(data)
    max_examples = examples - lookback * input_steps + 1 - forecast_step - forecast_len␣
↪+ 1
    if max_examples <= 0:
        raise ValueError(f"No examples generated from data of shape {data.shape} with␣
↪lookback {lookback}, input_steps {input_steps}, forecast_step {forecast_step},␣
↪forecast_len {forecast_len}")

    x = np.empty((max_examples, lookback, num_inputs), dtype=np.float32)
    y = np.empty((max_examples, forecast_len, num_outputs), dtype=np.float32)

    for i in prange(max_examples):
        x_start_idx = i
        x_end_idx = x_start_idx + lookback * input_steps
        y_start_idx = x_end_idx + forecast_step - input_steps
        y_end_idx = y_start_idx + forecast_len

        x[i] = data[x_start_idx:x_end_idx:input_steps, :num_inputs]
        y[i] = data[y_start_idx:y_end_idx, -num_outputs:]

    y = np.transpose(y, (0, 2, 1)).astype(np.float32)

    return x, y
```

```
data = np.random.rand(5000, 6).astype(np.float32)  # Dummy data
lookback = 365
num_inputs = 5
num_outputs = 1
N = 1000
```

```
start = time.time()
for i in range(N):
    x, y = prepare_data(data, np.int32(lookback), num_inputs, num_outputs, np.int32(1),␣
↪np.int32(0), np.int32(1))

print("Time taken :", time.time()-start)
```

```
Time taken : 67.57351446151733
```

```
start = time.time()
for i in range(N):
    x, y = prepare_data1(data, np.int32(lookback), num_inputs, num_outputs, np.int32(1),␣
↪np.int32(0), np.int32(1))
print("Time taken with simple numba:", time.time()-start)
```

```
Time taken with simple numba: 15.34597110748291
```

```
start = time.time()
```

(continues on next page)

(continued from previous page)

```
for i in range(N):
    x, y = prepare_data2(data, np.int32(lookback), np.int32(num_inputs), np.int32(num_
↪outputs), np.int32(1), np.int32(0), np.int32(1))
print("Time taken type annotation with numba:", time.time()-start)
```

```
Time taken type annotation with numba: 15.80981731414795
```

```
start = time.time()
for i in range(N):
    x, y = prepare_data3(data, lookback, num_inputs, num_outputs, np.int32(1), np.
↪int32(0), np.int32(1))
print("Time taken using prange in numba:", time.time()-start)
```

```
Time taken using prange in numba: 16.917538166046143
```

Total running time of the script: (2 minutes 3.935 seconds)

7.10 Scripts to executables

```
import os
import sys
from collections import Counter


def main(path, start:int=26):

    if not os.path.exists(path):
        print(f"{path} does not exist.")
        return

    stations = [file[start:] for file in os.listdir(path)]
    uniques = set(stations)

    # find duplicates in stations
    counts = Counter(stations)
    duplicates = {item:count for item, count in counts.items() if count > 1}
    if duplicates:
        print(f"Found {len(duplicates)} duplicates in {path}")
        for duplicate, count in duplicates.items():
            print(f"{duplicate} found {count} times")

        print(f"There are {len(uniques)} unique files.")
    else:
        print(f"No duplicates found in {path}")
```

```
if __name__ == "__main__":

    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    start = int(sys.argv[2]) if len(sys.argv) > 2 else 0
    main(path, start)
```

```
No duplicates found in /home/docs/checkouts/readthedocs.org/user_builds/python-seekho/
↪checkouts/latest/scripts/advanced
```

pyinstaller –onefile scripts_to_executables.py

dist/scripts_to_executables

dist/scripts_to_executables /mnt/datawaha/hyex/atr/gscad_database/raw/BOMAustralia/zip_files

dist/scripts_to_executables /mnt/datawaha/hyex/atr/gscad_database/raw/BOMAustralia/zip_files 26

Total running time of the script: (0 minutes 0.002 seconds)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search