



# PYTHON

a python  
programming  
tutorial from zero  
to mastery

Mohammad  
Montazeri

PythOn to 100  




*This page intentionally left blank*

# Preface

This note book has been documented with this purpose to provide all information required for an absolute amateur Python learner to alter into a pro programmer. It is severely suggested for beginners to learn essential basics of programming and algorithm design before starting this course. That's because intermediate programming concepts are explained here briefly and can be understood better with previous knowledge. C++ can be a good start to enter programming world. Fortunately, Fundamentals of Computer and Programming has been a mandatory course of most engineering schools lately, and that provides sufficient knowledge to use this tutorial.

There are some frequently used fonts in this note that indicate special characteristics of their content. it is necessary to know some important ones for better comprehension:

```
this font is used for codes in IDE, especially  
for providing examples
```

after the latter font, usually comes this one as the result of running the code.

*This also is used for small pieces of codes brought within context.*

***This might also be found as pieces of codes, mostly used in Terminal or CMD.***

In some cases, the result of our code is a figure; the figure is attached after the code in these cases.

In addition to fonts, there are some very repeated abbreviations in this notebook, the user would better be familiar with before:

e.g. → example

var → variable

dict → dictionary

func → function

err → error

char → character

attr → attribute

sym → symbolic

expr → expression

p.s. → postscript



# Acknowledgments

The author of this notebook, Mohammad Montazeri, is a student of Mechanical Engineering at University of Tehran, Iran. This notebook represents the result of two summers dedicated to gathering the book.

The author's first acquaintance with Python and programming took place at Fundamentals of Computer and Programming course taught by professor Jamal Kazazi in College of Engineering. With his guidance, this path emerged ahead of the author to step in the route of programming.

To gather this documentation, the author was blessed with helps of Mohammad Ordookhani<sup>1</sup> and Amirhossein Bigdelu<sup>2</sup> via their website platforms Toplearn<sup>3</sup> and Mongard<sup>4</sup>. For this, the author is sincerely grateful to them and all who have inspired him in his path.

**Mohammad  
Montazeri**

[mohammadmontazeri313@gmail.com](mailto:mohammadmontazeri313@gmail.com)  
<https://t.me/MohammadMontazeri>

---

<sup>1</sup> <https://ordookhani.com/>

<sup>2</sup> <https://github.com/amirbigg>

<sup>3</sup> <https://toplearn.com/c/o2V3>

<sup>4</sup> <https://www.mongard.ir/courses/python-beginner-course/>

# Table of contents

Introduction .....	1
Getting Started .....	2
Basics .....	6
Logical Operators and IF Statements .....	7
Loops .....	8
Variables and Data Structures .....	9
Lists .....	11
Slicing .....	14
List Comprehension .....	14
Nested Lists .....	15
Tuples .....	16
Sets .....	16
Union and Intersection .....	17
Dictionaries .....	18
Dictionary Comprehension .....	21
General Useful Functions .....	22
Input and Output .....	28
Print .....	28
Formatting the Output .....	28

Precision Handling .....	29
Input .....	30
Functions .....	31
Star Args (*args) .....	32
Key Word Args (**kwargs) .....	33
Lambda .....	35
Maps .....	36
Filter .....	36
Errors .....	38
Error Handling .....	38
Error Raising (Throwing) .....	40
Bubble Sort .....	41
Python Debugger (PDB) .....	43
Object Oriented Programming .....	44
Self .....	45
Initial Functions .....	45
Name Mangling .....	46
Class Attributes .....	47
Class Methods .....	48
Representation .....	50
Property .....	50
Inheritance .....	53

Super .....	54
Multiple Inheritance .....	57
Method Resolution Order .....	59
Polymorphism .....	62
Dunder Methods .....	63
Iterator .....	65
Generator .....	69
Decorator .....	72
Decorator Factory .....	77
File .....	79
Modules .....	81
Numpy .....	84
Matrix .....	84
Useful Methods .....	85
File .....	86
Sympy .....	88
Definition of Symbolic Variables .....	88
Substitution of Values .....	88
Showing the Results .....	89
System of Equations .....	89
Differentiation .....	91
Plot .....	92



Scipy .....	94
Matplotlib .....	95
Plot .....	96
Useful Customization Methods .....	97
Color Demo .....	101
r-strings .....	105
Multiple Plots .....	107
Bar Graphs and Histograms .....	109
Scatter Plots .....	111
Stack Plots .....	113
Pie Charts .....	114
Time Series .....	116
Live Plots .....	119
Other Plots .....	121
Csv .....	126
Pandas .....	129
Minor Modules .....	130
Time .....	130
Termcolor .....	130
Pyfiglet .....	130
Datetime .....	131
Math .....	131

Virtual Environment .....	132
Graphical User Interface .....	134
Appendix .....	145
References .....	150

# Introduction

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.<sup>1</sup>

Python syntaxes are so close to English language, thus, easy to learn. According to [Stack Overflow Developer Survey](https://insights.stackoverflow.com/survey/)<sup>2</sup>, Python has been one of the most popular languages of late years worldwide. Python is also loved by data scientists and used a lot in machine learning (ML) and artificial intelligence (AI).

Python is open source, widely applicated, user friendly and easy to learn. It has also an enormous community and huge variety of libraries to help programmers develop their codes easily and quickly.

Like any other language, Python has its disadvantages too. Python is slow, compared to some other popular languages such as C++ or Java, since it uses interpreter instead of compiler; It also consumes more memory and can't compete with them on Mobile Phone programming.

---

<sup>1</sup> [What is Python? Executive Summary | Python.org](https://www.python.org/doc/essays/blurb/) <https://www.python.org/doc/essays/blurb/>

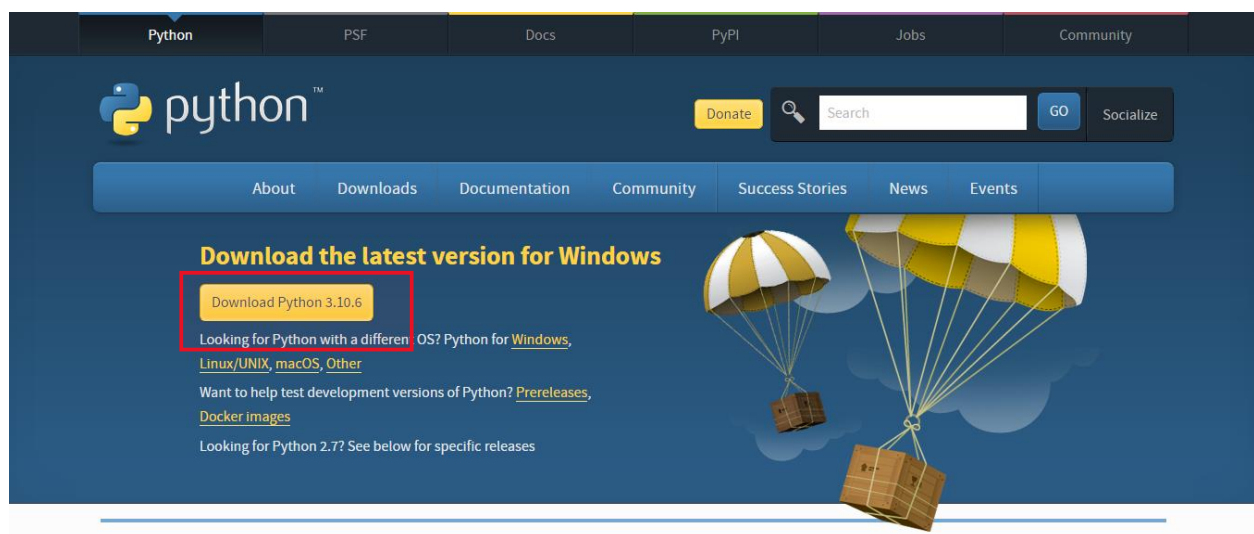
<sup>2</sup> <https://insights.stackoverflow.com/survey/>

# Getting Started

To set up Python programming vitals, you need Python Interpreter core for running scripts and an IDE (text editor) as your coding environment.

To install Python Interpreter, visit its official website [Welcome to Python.org](https://www.python.org/)<sup>1</sup>.

On the Downloads tab, download latest version of Python compatible to your operation system.

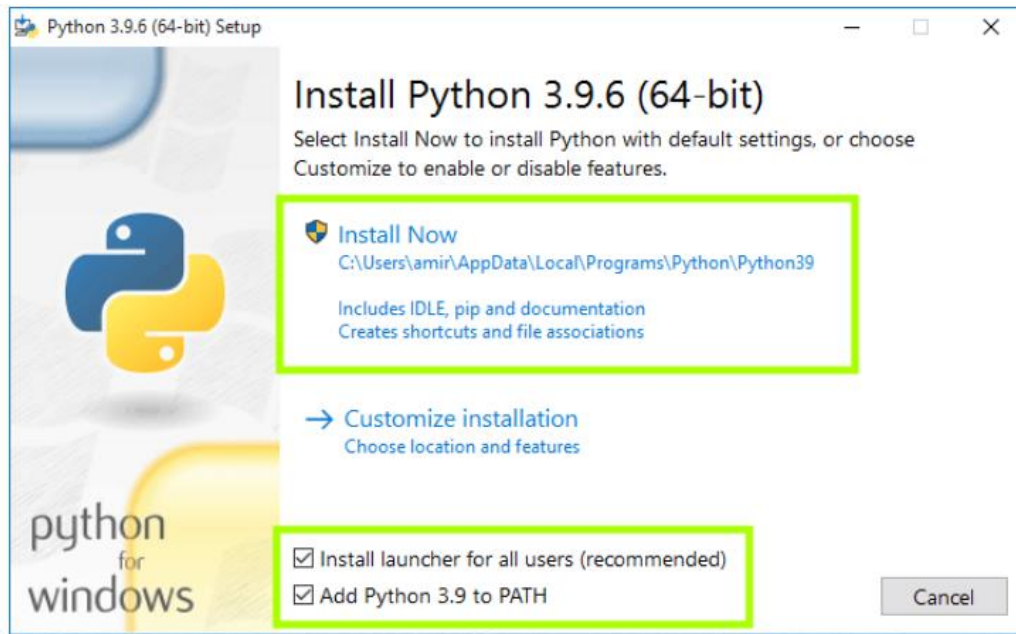


After downloading process, open up the installation *exe* file.

While installing, don't forget to mark both options below:

---

<sup>1</sup> <https://www.python.org/>




After installation process, you are ready to go. But to ensure successful installation, open up systems Terminal or command prompt and enter '**python**'. You must see the something like the following result:

```
C:\Users\Mohammad>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To make sure Python's *pip* tool is sound and ready to work, open up CMD once more, and type '**pip --version**'. Then you must be able to see *pip*'s version.

You can call Python from every directory, since you've checked the box for adding Python to path.

Now you need a text editor for your codes to be born on. Any text editor might do fine; from simple notepad++ or vscode, to enormous visual studio and such. But the most suggested IDE dedicated to Python programming is PyCharm. To download it, visit its official website [PyCharm: the Python IDE for Professional Developers by JetBrains](https://www.jetbrains.com/pycharm/)<sup>1</sup>.



Version: 2022.2.1  
Build: 222.3739.56  
17 August 2022

[System requirements](#)  
[Installation instructions](#)  
[Other versions](#)  
[Third-party software](#)

### Download PyCharm

[Windows](#) [macOS](#) [Linux](#)

#### Professional

For both Scientific and Web Python development. With HTML, JS, and SQL support.

[Download](#)


Free 30-day trial available

#### Community

For pure Python development

[Download](#)

Free, built on open-source



Get the Toolbox App to download PyCharm and its future updates with ease

You can easily download Community version of PyCharm for free.

Also, [colab.research.google.com](https://colab.research.google.com/)<sup>2</sup> can be used as an online Python IDE as well.



<sup>1</sup> <https://www.jetbrains.com/pycharm/>

<sup>2</sup> <https://colab.research.google.com/>



Here is a little list of useful shortcuts in PyCharm:

Shift+F10 → run last compiled file

Ctrl+Shift+F10 → compile and run current file

Ctrl+D → duplicate line or selection

Ctrl+W → expand selection

Ctrl+Shift+W → shrink selection

Ctrl+Alt+L → organize your codes (make it neat)

Ctrl+/ → comment or uncomment line(s)

Ctrl+Shift+arrow keys → moving lines upward or downward

Ctrl+E → recent files

Ctrl+R → find and replace

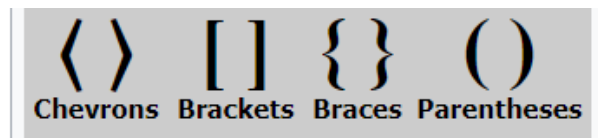
# Basics

In Python, nothing is used to end a line of code, despite some other languages like C++.

Use hashtag '#' at the first of a line or phrase to declare it as a comment.

Use three *single-quotation-marks* (') at the first and the end of a section to comment whole section. This kind of comments are called **docstrings**. Unlike simple comments \_which are usually meant to guide the programmers of THE code\_, docstrings are mostly used to give a summary of what our code does, and how it does it, to the public.

use TAB for declaring the boundaries of classes, functions, etc. (instead of BRACES in C++). this TABs are called indents.



Like C++ we have the syntax `a += 1` as a short form of `a = a + 1` but we don't have `a++`. other mathematical operands, too, can be used with this method.

If a line is too long, continue it on the next line, by putting a backslash '\ ' at the end of the main line. Check out this site for further info: [Write a long string on multiple lines in Python | note.nkmk.me](https://note.nkmk.me/en/python-long-string/)<sup>1</sup>

---

<sup>1</sup> <https://note.nkmk.me/en/python-long-string/>

# Logical Operators and IF Statements

we can use multiple ways for if-else sentences:

```
if condition:
```

```
    command
```

```
elif condition:
```

```
    command
```

```
else:
```

```
    command
```

```
if ... is ...: command
```

```
if ... is not ...: command
```

```
command if condition else command
```

LOGICAL OPERATORS in Python are: '***and***', '***or***', '***not***'.

Like C++ use '==' or '***is***' for equality and '!=' or '***is not***' for inequality; but there is a little difference between '***is***' and *double equation*:

Actually, '==' returns True if two variables have equal values. But '***is***' returns True if two variables have equal values AND they refer to THE SAME reference (they should point to the same spot in memory).

# Loops

```
while condition:  
    commands  
    break
```

```
for iteration:  
    commands  
    break
```

E.g.:

```
for i in range(6):  
    print(i*' '*')
```

```
*  
**  
***  
****  
*****
```

(p.s.: you'll learn more about [range\(\)](#) a little later in headline 'Lists')

[while True](#) is used for infinite loops and [break](#) is usually used in if statements to end these loops.

Another command used in loops, is [continue](#). This command, too, is usually used with an if statement, so that if the condition is granted, using this command, you'll be able to skip one cycle of the loop by ignoring the rest of the lines in the loop in that specific cycle.

[True](#) and [False](#) in Python are written with their first letter being capital.

Operator `'//'` returns INTEGER part of a division; e.g.: `22//10 = 2`

**Power** is denoted with `**` in Python; e.g.: `2**3 = 8`

# Variables and Data structures

Some languages like C++ and C# are statical languages, meaning that a variable supports only one type; Python, however, is a dynamical language where a variable can change its type. It means if for example you've defined a var as integer, you're able to assign other types such as string to it in other lines.

These are the most common Python variable types:

- Int `x = 1`
- Float `x = 1.`
- Str `x = 'hi'`
- List `x = [1, 2, 3]`
- Tuple `x = (1, 2, 3)`
- Set `x = {1, 2, 3}`
- Dictionary `x = {'a' : 1, 'b' : 2}`
- Bool `x = True`
- Nonetype `x = None`

A way to give value to multiple variables at once:

`x, y = 1, 2`      or      `x = y = z = 1`

There is no difference between single and double quotation (" " / ' ' ).  
but if you want to use quotation inside a string, it should be declared  
with different kind of quotation mark; e.g.: `sentence = "he said: 'hello' "`  
/or/ `'he said: "hello" '`

In addition, we use backslash before special characters like `\n` or `\t`.<sup>1</sup>

We can use '+' to add strings to the end of each other; also, the method `str1 += str2`.

Strings' index starts from zero (same as lists).

Use `.upper()` and `.lower()` to capitalize all chars of a string and vice versa. E.g.:

```
x = 'hi babe'
y = x.upper()
print(y)
```

HI BABE

And replace a word in a string by `.replace('old word', 'substitute word')`

```
y = x.replace('hi', 'bye')
print(y)
```

bye babe

Best methods for showing strings with desired formats are listed in 'Input & Output' headline.

To convert data types, see this example: `x = 123`    `y = float(x)`

/or/ `x = "1234"`    `y = int(x)`

`round(a, b)` → rounds the number 'a' to 'b' digits. You can use it this way too: `a.round(b)`

---

<sup>1</sup> Find out a full list of these chars in the Appendix.



zero, empty strings or objects, and *None* vars have **False** value in Boolean space.

**Iterable**, in the simplest definition, is a variable with multiple indexes, like lists, tuples, strings, etc.

You can **delete** a var such as 'a' from your workspace like this: `del a`

## Lists

Lists are the same as *arrays* in C++. Every segment of lists can have a different type in spite of C++. To access the segments, we do as in C++:

`a = [1, 2, 3, 4]`      `a[0] = 1`      `a[3] = 4`

To access the segments from end, we use negative numbers; also, we can decrease this number to move backwards in the list.

`a[-1] = 4`      `a[-2] = 3`

See these examples to understand better about iteration in lists:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[1:8:3]) # [start index : stop index : step]
print(a[2::2])  # stop undeclared, means up to the end
```

[2, 5, 8]

[3, 5, 7, 9]

```
a = [1, 2, 3, 4]
for number in a:
    print(number*2)
```

2

4

6

8

**range(a, b, c)** → creates an order of INTEGER numbers starting from 'a' (the 'a' itself included) to 'b' (the 'b' not included) with step 'c'. this function doesn't make a LIST; to do so, we have to convert it to list by ourselves: `list(range(a, b, c))`

**range(b)** → starts from 0 to a-1

(While not assigned, default value of *a* and *c* is respectively 0 and 1)

**len(a)** → shows the length of list 'a'. also works like: `a.__len__()`

to see if a specific value exists in our list → the order `a in b` returns a Boolean; True if value 'a' is in list 'b', and False if it's not.

If we add two lists using *plus* operand '+', it will add second list's items to the end of first list. if we have two lists of numbers and want to sum up their corresponding items, we should use some additional libraries like *Numpy* which will be explained later in the section *Modules*.

To add only one item to our list → `ourList.append(item);`

But to add multiple items → `ourList.extend(items).`

These 2 methods add item(s) in the end; but for intending a specific place to add our item to → `ourList.insert(index, item).`

Now to delete all items in a list → `ourList.clear()`

`ourList.pop(index)` → deletes the mentioned index. If you don't give a number to this method, it deletes last item; and if you write:

`variable = ourList.pop(index)` → in addition to what's been said, it saves the deleted item in THE variable.

We can delete a particular segment of our list by using its index via this way too → `del ourList[index]`

**ourList.remove(item)** → removes THE FIRST segment whose value is THE item.

**ourList.index(item)** → returns a number which is the first index dedicated to THE item (counting from zero). you can add Start & Stop values after item too.

**ourList.count(item)** → counts how many segments exist in the list with the value 'item'.

**ourList.reverse()** → updates our list by reversing it.

**reversed(ourList)** → in spite of last method that applies changes to THE list, this one saves it in a variable.

**ourList.sort()** → updates our list by sorting its contents from small to big (or alphabetically if they're strings). There is a more advanced form of this order known as 'sorted' that will be explained after introducing all kinds of variables.

**'a'.join(ourList)** → creates a string which has the values of our list, separated with char 'a'. Note that all values in our list must be string for this job. See the example:

```
x = ['a', 'b', 'c', 'd']  
y = '**'.join(x)  
print(y)
```

```
a**b**c**d
```

## SLICING

Imagine we have a list like `a = [1, 2, 3, 4, 5, 6, 7, 8, 9]`. Now if we say `b = a[1]`, `b` would be an integer with value '2'. But if we say `b = a[1:6:1]`, `b` will be a list based on `a`, with value `[2,3,4,5,6]`.

this method is called slicing. in mentioned syntax, we will have arguments in this sequence: `[start : end : step]` (note that the segment of the 'end' index will not be included)

we can spare ourselves some of these arguments; e.g.:

`list2= list1[:]` → copies all values of list1 into list2. We can use negative arguments as well; for example `x = x[::-1]` is equal to `x.reverse()`. Since strings are a kind of lists, this method can be used for them too.

## LIST COMPREHENSION

We can operate on values of a list to make a new one:

```
a = [1, 2, 3, 4]
b = [n * 2 for n in a]
print(b)
[2, 4, 6, 8]
```

We can also have this for strings, as well as we can have logical operators:

```
a = 'hello world' # to extract vowel letters:
b = [ch.upper() for ch in a if ch in ['a', 'e', 'i', 'o', 'u']]
print(b)
['E', 'O', 'O']
```

in order to use both *IF* and *ELSE*, we should mention the logical clause before *FOR* clause:

```
a = [1, 2, 3, 4, 5, 6] # to extract EVEN numbers
b = [n if n%2 == 0 else 0 for n in a]
print(b)
```

[0, 2, 0, 4, 0, 6]

## NESTED LISTS

Multi-dimensional arrays are called nested lists.

```
a = [ [0,1,2] , [3, 4, 5] ]
print(a[0] , a[0][0], a[1][2])
```

[0, 1, 2] 0 5

Tricks we use to print nested lists:

```
a = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
for n1 in a:
    for n2 in n1:
        print(n2, end=' ')

# or using list comprehension:

print()
b = [[print(N2, end=' ') for N2 in N1] for N1 in a]
```

1 2 3 4 5 6 7 8 9 10 11 12  
1 2 3 4 5 6 7 8 9 10 11 12

# Tuples

Tuples are immutable lists; meaning that their values cannot be changed or modified after creation. Besides, they're faster than lists, because they occupy less storage.

Tuples are like lists in other aspects. E.g.:

```
myTuple = (1,2,3)      /or/      myTuple = tuple([1,2,3])
```

We can access to tuple items just like lists. Furthermore, the Slicing and Comprehension methods are doable for tuples too. In other words, most functions and methods that work with lists, can work with tuples as well.

# Sets

Sets are list of objects identified by braces {}. Sets don't have index numbers (to access their items), don't have repeated items (repeated items all count as one item). Sets are created like: *mySet = {1,2,3,'hi'}*. also, no order is applied to their segments while being printed. Like other iterables, sets, too, can be used as iteration factor in loops FOR and WHILE. They can be transformed to a list and vice versa. Some important functions for sets:

- *mySet = set(myList)*
- *ourSet.add(item)*
- *ourSet.copy()*



- `ourSet.clear()`
- `ourSet.remove(item)` → if item didn't exist, causes ERR
- `ourSet.discard(item)` → if item didn't exist, doesn't cause ERR.

Like all iterables, to see if an item exists in the set, this syntax *item in mySet* returns a Boolean. In addition, Comprehension method works here too.

## UNION AND INTERSECTION

There are these two operators '|' and '&' that are only for sets and work as union of sets and their intersection. In addition, there are these functions *.union()* and *.intersection()* as available equivalents too. Also, the function *.issubset()* checks if a set is a subset of another. E.g.:

```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7}
c = {3, 4}
print(a | b, a & b)
print(a.union(b), c.issubset(b))
```

```
{1, 2, 3, 4, 5, 6, 7} {3, 4, 5}
{1, 2, 3, 4, 5, 6, 7} True
```

# Dictionaries

As an equivalent of Structs in C++, we have dictionaries in Python. Dicts are identified by braces like sets, however, their difference is they have both *Keys* and *Values*, while sets have pure values only. There are two ways to define a dict:

```
a = { 'key' : value , 'key' : value , 'key' : value }
```

```
a = dict( key = value , key = value , key = value )
```

we access dict items using their keys like `ourDic['key']`; we can also use the order `ourDic.get('key')` as well. The difference is, in latter method, if the KEY doesn't exist in `ourDic`, it returns the value *None*, but the former method causes ERR in this case.

Some commands like `.values()` , `.keys()` and `.items()` are useful functions for dicts too. E.g.:

```
a = { 'name': 'Tom', 'age': 12, 'weight': 50 }
b = dict(name='Mads', age=19, weight=80)

print(b["name"])
print(a.keys())
print(a.values())
print(b.items())
print('-----')
for values in a.values():
    print(values)
print('-----')
for key, value in b.items():
    print(key, value)
print('-----')
for everything in a.keys():
    print(everything, '\t:\t', a[everything])
```

```
Mads
dict_keys(['name', 'age', 'weight'])
dict_values(['Tom', 12, 50])
dict_items([('name', 'Mads'), ('age', 19), ('weight', 80)])
```

```
-----
Tom
12
50
```

```
-----
name Mads
age 19
weight 80
```

```
-----
name :      Tom
age  :      12
weight :    50
```

to see if something exists in our dict:

```
a = dict(name='Mads', age=19, weight=80)

print('name' in a)
print('Mads' in a)
print(19 in a)
print(19 in a.values())
```

```
True
False
False
True
```

The order `ourDic.clear()` clears all inside `ourDic`.

The order `ourNewDic = ourDic.copy()` copies `ourDic`'s items in `ourNewDic`.

To make a dict that all its values are the same, and with Keys 'keyList', and values 'val', the order `dict.fromkeys(keyList, val)` or `{}.fromkeys(keyList, val)` does the job so easily. E.g.:

```
a = {}.fromkeys(['a', 'b', 'c', 'd'], 1)
b = dict.fromkeys('abcd', 0)
print(a, '\t\t', b)
```

{'a': 1, 'b': 1, 'c': 1, 'd': 1}

{'a': 0, 'b': 0, 'c': 0, 'd': 0}

`ourDic.pop('key')` → deletes the mentioned *key* and its value. if you write `a = ourDic.pop('key')`, in addition to what's been said, it saves the **value** of deleted *key* in 'a'. `ourDic.popitem()` does the same with the LAST KEY.

The order `ourDic.update('elseDic')` adds the *elseDic*'s items to *ourDic*. If an item in *ourDic* has the same KEY as in the *elseDic*, it gets overwritten (updated).

`ourDic['Key'] = newValue` → To update only one item in *ourDic* with a new value. If THE 'key' doesn't exist in our dict, it will add it to the dict.

We can use tuples as dictionaries' keys as well. Also, we can turn a list of tuples into dictionary. E.g.:

```
locations = {
    (11.24, 32.6): 'Tehran',
    (39.89, 14.73): 'Sari'
}
height = dict([('Alvand', 1000), ('Sahand', 900)])
print(locations[(11.24, 32.6)], height)
```

Tehran {'Alvand': 1000, 'Sahand': 900}

# DICTIONARY COMPREHENSION

Same as lists, we can have the Comprehension method for creating dicts, like this:

```
# too long lines have been continued in their
# next line; that's the reason of backslashes
# at the end of some lines

a = dict(name='Jane', age=20, lessons= \
        ['statics', 'calculus', 'french'])

b = {keys: values for keys, values in a.items()}

c = {keys: (values if type(values) is not list \
        else 'unknown') for keys, values in b.items()}

d = {keys: values for keys, values in b.items() \
        if type(a[keys]) in [str, int]}

print(a, b, c, d, sep='\n')
```

```
{'name': 'Jane', 'age': 20, 'lessons': ['statics', 'calculus', 'french']}
{'name': 'Jane', 'age': 20, 'lessons': ['statics', 'calculus', 'french']}
{'name': 'Jane', 'age': 20, 'lessons': 'unknown'}
{'name': 'Jane', 'age': 20}
```

Another more complicated example:

```
a = {key: val for key, val in [(1, 4), (2, 5), (3, 6)]}
b = {key: (val if val < 27 else 'beyond maximum') \
        for key, val in [(i, 2 * i) for i in range(10, 16)]}
print(a, b, sep='\n')
```

```
{1: 4, 2: 5, 3: 6}
{10: 20, 11: 22, 12: 24, 13: 26, 14: 'beyond maximum', 15: 'beyond maximum'}
```

# General Useful Functions

The following is some general functions used for almost all iterables:

**all()** → If all items in an iterable are true, or, the iterable is empty, returns True. Otherwise, False.

**any()** → If at least one item in an iterable is true, returns true. Otherwise, false.

```
a = [0, 1, 2, 3]
b = [1, 2, 3, 4]
c = [0, 0, 0, 0]
d = []

tot = [a, b, c, d]
for x in tot:
    print(x)
    print('all :', all(x))
    print('any :', any(x))
    print('-----')
```

[0, 1, 2, 3]

all : False

any : True

-----

[1, 2, 3, 4]

all : True

any : True

-----

[0, 0, 0, 0]

all : False

any : False

-----

[]

all : True

any : False

-----



**sorted()** → as an advanced form of function `.sort()` introduced in 'Lists' title, this func, sorts the iterables with more options, like *reverse*, or sorting dicts based on their keys. E.g.:

```
a = [1, 2, 6, 9, 0, 89, 45]
b = sorted(a, reverse=True)
print(b)
```

[89, 45, 9, 6, 2, 1, 0]

```
cars = [
    {'brand': 'Porsche', 'model': '911', 'price': 2000},
    {'brand': 'BMW', 'model': '640li', 'price': 1000},
    {'brand': 'Volvo', 'model': 'xc40', 'price': 900},
    {'brand': 'Audi', 'model': 'eTron', 'price': 2200}
]
a = sorted(cars, key=lambda n: n['brand'])
b = sorted(cars, key=lambda n: n['price'], reverse=True)
c = sorted(cars, key=lambda n: n['model'])
# you'll learn more about 'lambda' in header 'Functions'

d = [a, b, c]
for n in d:
    for m in n:
        print(f'{m["brand"]} \t {m["model"]} \t {m["price"]}')
    print('-----')
```

```
Audi   eTron  2200
BMW    640li  1000
Porsche      911   2000
Volvo   xc40   900
```

```
-----
Audi   eTron  2200
Porsche      911   2000
BMW    640li  1000
Volvo   xc40   900
```

```
-----
BMW    640li  1000
Porsche      911   2000
Audi   eTron  2200
Volvo   xc40   900
```

**max()** & **min()** → to find maximum and minimum segment of an iterable. For dicts and complicated cases, use a *key*. For example, in resume of last example, we can have:

```
a = min(cars, key=lambda x: x['brand'])
b = min(cars, key=lambda x: x['price'])

c = ['R8', 'LP850', 'gt', 'z11']
d = max(c)
e = min(c, key=lambda x: len(x))

print(a, b, d, e)
```

```
{'brand': 'Audi', 'model': 'eTron', 'price': 2200} {'brand': 'Volvo', 'model': 'xc40', 'price': 900} z11
R8
```

**sum()** & **round()** → as their names refer, do the numerical jobs as this example:

```
a = [1, 2, 3, 4, 5]
print(sum(a), sum(a, 10))

b = [4.4, 4.5, 4.6]
c = [print(round(n)) for n in b]

d = 4.3456
print(round(d, 2), round(d, 3), round(d, 7))
```

```
15 25
4
4
5
4.35 4.346 4.3456
```

**zip()** → gets multiple iterables as arguments and pairs up their items. Oppositely, it can make up multiple tuples from an iterable whose contents are paired. Zip's results are iterable just once; meaning that

when you call the variable in which zip has been saved, for the next time, this variable will have no data inside. In other words, its contents get wiped out after once used. To unzip an iterable, we use a star '\*' before mentioning its name as argument. See the examples and note the comments for instructions and hints:

```
# zipping data:
a = [1, 2, 3, 4]
b = [2, 4, 6, 8]
name = ['Jared', 'Jake', 'Jane', 'John']

c = zip(a, b, name)
d = zip(name, b)

# backslashes used to continue too long lines in their
# next line
print(c, list(c), tuple(c), dict(d), sep='\n', \
      end='\n-----\n')

# -----
# on the contrary --> unzipping data
e = [(1, 2), (2, 4), (3, 6)]
f = (('Jane', 1, 2), ('John', 2, 4), ('Jack', 3, 6))

print(list(zip(*e)), set(zip(*f)), sep='\n', \
      end='\n-----\n')

# -----
# the number of pairs is equal to the least number
# of contents among iterables.
# like MAPs, zips are iterable just once;
# thats why if you print a var made with a zip twice,
# it'd be empty the 2nd time
g = [1, 2, 3]
h = [2, 4, 6, 8, 10]
i = zip(g, h)
print(list(i), '\t', dict(i), end='\n-----\n')

# -----
# imagine that we have a bunch of students,
```

```

# and we want to determine their average score
# in midterm and final exam;
# also their maximum score between these 2 quizzes:

students = ['Jaimie', 'Jon', 'Jerald']
midterm = [19, 16.25, 14.5]
finalExam = [17, 18.5, 13.75]

maxScore = {whatever[0]: max(whatever[1], whatever[2]) \
             for whatever in zip(students, midterm,
                                finalExam)}
average = {whatever[0]: (whatever[1] + whatever[2]) / 2 \
           for whatever in zip(students, midterm,
                                finalExam)}

print(maxScore, average, '-----',
      sep='\n')

# another way to reach these results:

maxScore = zip(students, map(lambda x: max(x[0], x[1]), \
                                zip(midterm, finalExam)))
average = zip(students, map(lambda x: (x[0] + x[1]) / 2, \
                                zip(midterm, finalExam)))

print(dict(maxScore), dict(average), sep='\n')

```

```

<zip object at 0x0000022D6B498F40>
[(1, 2, 'Jared'), (2, 4, 'Jake'), (3, 6, 'Jane'), (4, 8, 'John')]
()
{'Jared': 2, 'Jake': 4, 'Jane': 6, 'John': 8}
-----
[(1, 2, 3), (2, 4, 6)]
{(2, 4, 6), (1, 2, 3), ('Jane', 'John', 'Jack')}
-----
[(1, 2), (2, 4), (3, 6)]  {}
-----
{'Jaimie': 19, 'Jon': 18.5, 'Jerald': 14.5}
{'Jaimie': 18.0, 'Jon': 17.375, 'Jerald': 14.125}
-----
{'Jaimie': 19, 'Jon': 18.5, 'Jerald': 14.5}

```

```
{'Jaimie': 18.0, 'Jon': 17.375, 'Jerald': 14.125}
```

**id(var)** → returns a number that is the address of the *var* in memory.

# Input and Output

## Print

We use the `print(objects)` function to display objects on the RUN window or TERMINAL. In case more than one object is supposed to be displayed, separate them with comma. If these objects are strings, and you want to attach them to the end of each other, you can use plus sign '+' instead of comma between them.

The advanced form of this function contains key words *sep* and *end* so that you can clarify what you want your objects be separated with on the display, and what should be printed after all of them have been shown: `print(*objects, sep = 'separating chars', end = 'ending chars')`

If these two keywords are not valued, they will have their default value: a space ' ' as *sep* and a new line '\n' as *end*. E.g.:

```
a = 3
print(1, 2, 'a ' + '=', a)
print(1, 2, a, 4, sep='**', end='\n -----')
```

```
1 2 a = 3
1**2**3**4
-----
```

## FORMATTING THE OUTPUT

The easiest and most common way to format and organize string-data compounds in output is using f-string. If you want to print an object **inside** a string, we put an 'f' before string and put the object(s) in braces (called space-holders). See the examples:

```
name = 'Hugh'
age = 40
print(f"your brother, {name}, says he's {age} years old")
```

your brother, Hugh, says he's 40 years old

yet there are other uncommon ways to do so. For instance, in resume of last example, we can have:

```
print("{} says he's {} years old".format(name, age))
print("{0} says he's {1} years old".format(name, age))
print("{1} says he's {0} years old".format(name, age))
print("{n} says he's {a} years old".format(n=name, a=age))
# old-fashioned C++ method:
print("%s says he's %d years old" % (name, age))
```

Hugh says he's 40 years old  
Hugh says he's 40 years old  
40 says he's Hugh years old  
Hugh says he's 40 years old  
Hugh says he's 40 years old

## PRECISION HANDLING

See the example to learn 3 ways of precision handling of float data:

```
# initializing value
a = 3.4536

# using "%" to print value till 2 decimal places
print("using '%' : ")
print('%.2f' % a)

# using format() to print value till 2 decimal places
print("using 'format' : ")
print("{0:.2f}".format(a))

# using round() to print value till 2 decimal places
print("using 'round' : ")
print(round(a, 2))
```

using '%' :  
3.45  
using 'format' :  
3.45  
using 'round' :  
3.45

## Input

To get values from the client working with the program, use `input(message)`

Inside parentheses, write what you want to be displayed on the monitor before getting input. E.g.: `a = input("enter a number")`

This function takes anything as string; to convert it to int or float act like: `b = int(a)` or `b = float(a)`

Also abbreviated in one line: `a = int(input())`

to intake multiple values → `input(message).split(splitter)`

in this method, client enters as many items as he wants, separated with the *splitter* char. E.g.:

```
a, b, c = input('enter 3 numbers to sum them up, ' \
               'separated with commas: ').split(',')
print(a + b + c)
print(int(a) + int(b) + int(c))
```

enter 3 numbers to sum them up, separated with commas: 2, 3, 5  
2 3 5  
10



# Functions

```
def function name(parameters):  
    commands  
    return results
```

You can ignore the *return* line if your function is not supposed to return anything. If you want your function to have some parameters with default values, add the value after an equality sign '=' in parameter list. In this case, if the mentioned parameter isn't valued when the function is being called, the default value gets used. In addition, you can assign values to parameters without their default order if you mention their **PARAMETER NAME** in your arguments when calling the function. E.g.:

```
def power(a, b=1):  
    print(a ** b)
```

```
power(3, 2)  
power(b=3, a=2)  
power(4)
```

```
9  
8  
4
```

Functions can take other functions as parameters or return them; there can also be a function inside another, but it won't be accessible outside the parent func, unless it's returned as a result. E.g.:

```
def starks(name, func):  
    print(f'{name} {func("stark")}')  
  
def Lordling(house):  
    return f'is from house {house}'  
  
starks('Bran', Lordling)
```

Bran is from house stark

There are also two other type of parameters used in functions:

## Star Args (\*args)

if you want your function to take multiple parameters, as many as the client wants, use a star '\*' before the parameter:

```
def math(opperand, *num):  
    if opperand == '+':  
        tot = 0  
        for n in num:  
            tot += n  
        print('summation = ', tot)  
    elif opperand == '*':  
        tot = 1  
        for n in num:  
            tot *= n  
        print('multiplication = ', tot)
```

```
math('+', 10, 20, 30)
math('*', 1, 2, 3, 4, 5, 6, 7)
```

```
summation = 60
multiplication = 5040
```

**note** that, when calling a func, using a LIST as the corresponding argument of a *\*arg parameter*, confronts ERR. To solve that, we should Unpack our LIST's items and pass THEM to the function. To do so, use a star before list's name as argument. E.g.:

```
def sum(*num):
    tot = 0
    for n in num:
        tot += n
    print(tot)

sum(10, 20, 30)
sum(*[10, 20, 30])
```

```
60
60
```

## Key Word Args (\*\*kwargs)

Use two stars **\*\*** before a parameter while defining a function to make it a *kwarg*. To assign value to such parameters, we should use both *key* and *value* as arguments while calling the func. Learn more of this type of parameters in the examples:

```
def func(**info):
    for key, value in info.items():
        print(f'{key} : {value}')
    print('-----')

func(name='Ford Raptor', price=1200, full_option=True)
func(name='Tom', lastName='Hardy', career='Actor')
```

```
name : Ford Raptor
price : 1200
full_option : True
-----
name : Tom
lastName : Hardy
career : Actor
-----
```

**note** that, when calling a func, using a DICTIONARY as the corresponding argument of a *\*\*kwarg parameter*, confronts ERR. To solve that, we should Unpack our DICTIONARY 's items and pass THEM to the function. To do so, use two stars before dictionary's name as argument. E.g.:

```
def func(**info):
    print(info)

d = {'name': 'Brad', 'age': '44', 'salary': 22000}
func(**d)
```

```
{'name': 'Brad', 'age': '44', 'salary': 22000}
```

If you want your function to have several types of parameters, you have to apply this sequence while defining parameters for the func:

1. Normal parameters
2. \*args
3. Parameters with default values
4. \*\*kwargs

## Lambda

Lambda is mostly used to define a simple and short function in one line. Also, lambda is a mediator to pass a func into another.

function name = lambda parameters: commands & results

e.g.:

```
sum = lambda a, b: print(a + b)
sum(10, 20)
print(type(sum))
print(type(sum(5, 10)))
```

```
30
<class 'function'>
15
<class 'NoneType'>
```

# Maps

Maps are called to functions that, as parameters, get a **lambda**, and at least another simple parameter.

```
variable name = map(lambda parameters: results , parameters)
```

It's an important attribute of maps that they're usable just once; in other words, after once called, maps get cleansed. See the examples for better understanding:

```
a = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, a)
print(squared)
print(list(squared))
print(list(squared))
```

```
<map object at 0x00000235AAC9BA60>
[1, 4, 9, 16]
[]
```

# Filter

Filters work like maps, mostly to filter the data of an iterable by lambda.

```
variable name = filter(lambda parameters: results according to  
conditions, parameters)
```

Filters, too, same as maps, are usable just once. See the example to find out more about filters:

```
a = [
    {'name': 'Ana', 'family': 'de Armas', 'age': 28,
     'appearance': 0},
    {'name': 'Ryan', 'family': 'Gosling', 'age': 34,
     'appearance': 1},
    {'name': 'Mark', 'family': 'Ruffalo', 'age': 44,
     'appearance': 1}
]

b = filter(lambda n: n['age'] >= 36, a)
print(list(b))
print(list(b))

# blend of Filter and Map:
c = map(lambda n: n['family'].upper(),
        filter(lambda m: m['age'] >= 30, a)
        )
print(list(c))

# the statement in front of the Lambda in Filter,
# returns a Boolean;
# if it's True, the item gets included in final result;
# if it's False, however, gets excluded.
# for example, we can find actors/actresses who DON'T
# have appearance in THE movie, using this method:
d = filter(lambda x: not x['appearance'], a)
print(list(d))
```

```
[{'name': 'Mark', 'family': 'Ruffalo', 'age': 44, 'appearance': 1}]
[]
['GOSLING', 'RUFFALO']
[{'name': 'Ana', 'family': 'de Armas', 'age': 28, 'appearance': 0}]
```

# Errors

*SyntaxError, NameError, TypeError, IndexError, ValueError, KeyError, AttributeError* and etc. are some of common errors we might face in python programming. Python compiler has specific explanations for each error that occurs and brings it after error type to help get rid of that: **NameError: name 'a' is not defined**

Note that error names are written just like above, with their first letter and the 'E' of 'Error' being capital. We need this exact spelling for handling and raising errors which we'll learn in resume.

## Error Handling

In order to keep our program flowing even if it faces errors, we use this method wherever needed:

```
try:
    commands
except (errName as varName):
    commands
else:
    commands
finally:
    commands
```



In this syntax, when there is no error for running try commands, it will run successfully as before. Otherwise, except commands occur. else and finally are optional and can be not mentioned at all. else commands occur only if the try section had worked; finally commands, on the other hand, run anyway. You can have multiple excepts for different types of errors; determine the error type in this case for every except (don't forget proper spelling). If you'd sooner use the explanation Python compiler provides for the err, you can save it in a variable if you add the 'as varName' part after error name in excepts. E.g.:

```
a = '12'
try:
    print(int(a))
except:
    print('there\'s been an err:')

b = '12s'
try:
    print(int(b))
except:
    print('there\'s been an err.')
else:
    print('no error occurred')
finally:
    print('end')

try:
    print(int(c))
except ValueError as x:
    print('ValueError', x)
except NameError as x:
    print('NameError:', x)
else:
    print('no error occurred')
finally:
    print('end')
```

```
12
there's been an err.
end
NameError: name 'c' is not defined
end
```

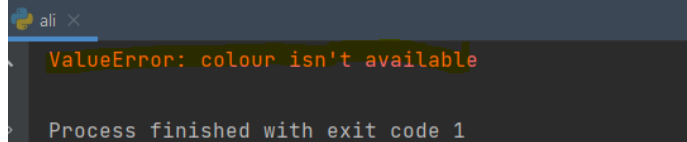
## Error Raising (Throwing)

Sometimes we intentionally create errors that if program went in an unintended direction, these errors stop it. Here is the syntax:

*`raise errName('explanation')`*

```
def coloured_printing(text, colour):
    availables= ['red', 'blue', 'black']
    if type(text) is not str:
        raise TypeError('text must be string')
    elif colour not in availables:
        raise ValueError('colour isn\'t available')
    else:
        print(f'{text} is printed in {colour}')

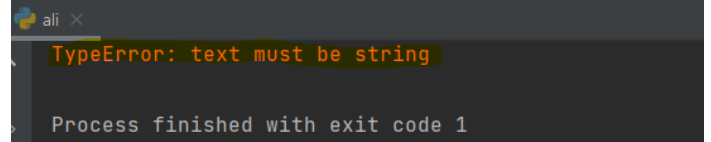
coloured_printing('hiiii', 'white')
```



The screenshot shows a terminal window with the command prompt 'ali'. The error message 'ValueError: colour isn't available' is displayed in red text. Below the error message, it says 'Process finished with exit code 1'.

```
def coloured_printing(text, colour):
    availables= ['red', 'blue', 'black']
    if type(text) is not str:
        raise TypeError('text must be string')
    elif colour not in availables:
        raise ValueError('colour isn\'t available')
    else:
        print(f'{text} is printed in {colour}')

coloured_printing(4, 'blue')
```

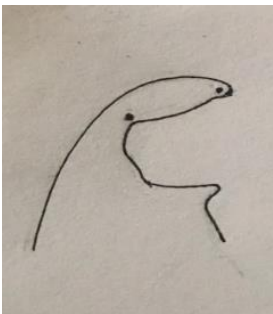


The screenshot shows a terminal window with the command prompt 'ali'. The error message 'TypeError: text must be string' is displayed in red text. Below the error message, it says 'Process finished with exit code 1'.

# Bubble Sort

As a practice of what's been said, it's worth mentioning Bubble Sort which is an important subject in coding. To learn that, consider this example that we have a 2D array, whose second dimension is a function of the first one. For example, 2<sup>nd</sup> list is first list + 1. In fact, we want to sort our data in first list from small to large, while keeping every item's correspondent value (in 2<sup>nd</sup> list) still attached to it (in the matter of sequence).

(p.s.: Author's feeling while trying to explain what's in his mind:)



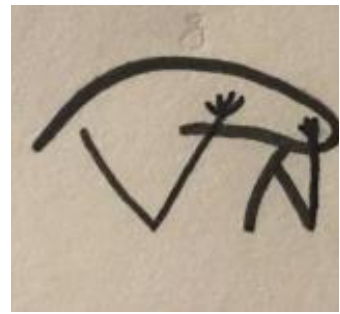
I



II



III



IV

```
# note: we use backslash '\' to continue long lines
# in their next line in this code :)

data = [[], []] # a 2D (nested) list

for n in [0, 1]: # taking value for both dimensions
    print(f'enter list-{n + 1} numbers, \
or any other key to finish this list.')
    while True:
        try:
            data[n].append(int(input()))
        except:
            break
```

```

print('entered lists are:', data[0], data[1], sep='\n')

def sorter(data):
    length = len(data[0])
    if length != len(data[1]):
        raise IndexError('Both lists must have \
the same length!')

    for i in range(length):
        for j in range(length - 1):

            if data[0][j] > data[0][j + 1]:
                data[0][j], data[0][j + 1] = \
                    data[0][j + 1], data[0][j]
                data[1][j], data[1][j + 1] = \
                    data[1][j + 1], data[1][j]

    return data

sorted_data = sorter(data)
print('sorted lists are:', sorted_data[0], \
      sorted_data[1], sep='\n')

```

enter list-1 numbers, or any other key to finish this list.	enter list-2 numbers, or any other key to finish this list.
1	12
24	12
06	6
17	39
5	10
6	8
3	1
g	h

entered lists are:

[1, 24, 6, 17, 5, 6, 3]

[12, 12, 6, 39, 10, 8, 1]

sorted lists are:

[1, 3, 5, 6, 6, 17, 24]

[12, 1, 10, 6, 8, 39, 12]

# Python Debugger (PDB)

PDB is a module (package)<sup>1</sup> we import to help us monitor the advance of lines running one after another. The function `set_trace()` controls the rest of the code after where it's been used. It works like a speed-bump that slows the flow of lines running continuously. In terminal, this method receives some commands:

- **N** → lets the next line run.
- **L** → shows the entire code in terminal and points which line we are on.
- **C** → continues running the rest of lines without interruption.
- **Var name** → if we have valued a var before calling `set_trace()` function, by typing its name in terminal, it shows us its value.
- **P** → if a var name is the same as a command of `set_trace()`, in order to see its value like above, we use this command before the var name.

Find more information at:

- <https://realpython.com/python-debugging-pdb/>
- <https://docs.python.org/3/library/pdb.html>

```
import pdb
pdb.set_trace()
```

(p.s.: you'll learn how to import packages later on. It's completely explained at headline '*Modules*'.)

---

<sup>1</sup> You'll learn about modules thoroughly later in the headline '*Modules*'.

# Object Oriented Programming

OOP is a very important subject in most programming languages. Here we learn most of its fundamentals in Python. There are two basic concepts in OOP, *class* and *object* (or instance). A class has *attributes* and *functionality*, and every object created based on the class, will have those attrs and funcs of its class.

Classes mostly have inner-functions called methods:

```
class student:
    name = ""
    std_num = 0

    def teacher(self):
        if self.std_num < 100:
            return "your class is upstairs"
        elif 100 <= self.std_num < 200:
            return "your class is downstairs"
        else:
            return "you don't have a class"

Fernand = student()
Fernand.name = "Fernand"
Fernand.std_num = 86
print("mr/ms {}, ".format(Fernand.name),
      Fernand.teacher())
```

mr/ms Fernand, your class is upstairs

# Self

when we're in a class and want to access its own attrs (variables defined inside the class), we use `self` as a metaphor of the object that is based on this class.

## Initial Functions

Some classes, get some data in the parentheses in front of the class's name when their object is being created. That's because of initial functions.

```
class car:
    price = 0

    def info(self):
        print(self.Brand + ' ' + self.Model \
              + ' in color ' + self.Color)
        if self.price <= 1000:
            print('is a cheap car')
        elif 1000 < self.price <= 5000:
            print('has a proper price')
        else:
            print('is an expensive car')

    def __init__(self, brand, model, color):
        self.Brand = brand
        self.Model = model
        self.Color = color

myCar = car('BMW', '740li', 'blue')
myCar.price = 3000
myCar.info()
```

BMW 740li in color blue  
has a proper price

## Name Mangling

There are some contractions for naming funcs and vars in Python that are distinguished via under score. For example:

- Name
- `_Name`
- `__Name`
- `__Name__`

The first way of naming objects is an ordinary way that we've used before. It is mostly known as **public** method. The second one, however, is used in classes or funcs for names that are NOT supposed to be SUGGESTED by the EDITOR (IDE) after typing only a DOT in front of an object's name. You access these kind of objects like: `object._Name`. this method may be known as **protected** in some languages.

The third way is the only one amongst these 4, that actually has some effects while coding. In classes, the function whose name is in this form, is known to be **private**; that's because, if you want to access them outside the class, it won't be suggested by IDE even if you type the full address like: `object.__Name`. To access these types, you should add the CLASS's name prefixed with an underscore before the name: `object._class__Name`. this method is called NAME MANGLING.

Finally, the last type doesn't differ from the first two, only it's more used to distinguish important or system-used names.



## Class Attributes

Imagine a class which has some variables inside. These vars will be the same for all of this class's objects, as long as we don't change them in a particular object. Fact is, these vars are common between the class and the instance; but if they're altered for a particular object, they will be separated in memory: one group for the class and other instances, and one for the mentioned object. E.g.:

(Note that the method `id(var)` returned a number which was the address of the `var` in memory. Also remember the difference between `is` and `==` in Python, explained in section *Basics*)

```
class site:
    users = ['Fernando', 'Goodwill', 'Leno']

    def __init__(self, name):
        if name in self.users:
            print('entrance succeeded')
        else:
            print('this username has not been
registered')

me = site('Goodwill')
you = site('Fernando')

print(site.users)
print(me.users)
print((you.users is me.users) & (site.users is me.users))
# if the last line shows True, it means that all three of
# class and its 2 instances have the attr <<users>>
# in the same spot in the memory

site.users = ['Alfred', 'Rose', 'Zack']
print(site.users, id(site.users))
print(me.users, id(me.users))
print(you.users, id(you.users))
```

```
me.users = ['Bernard', 'Suzan', 'Sasha']  
print(site.users, id(site.users))  
print(you.users, id(you.users))  
print(me.users, id(me.users))
```

```
entrance succeeded  
entrance succeeded  
['Fernando', 'Goodwill', 'Leno']  
['Fernando', 'Goodwill', 'Leno']  
True  
['Alfred', 'Rose', 'Zack'] 1888607607424  
['Alfred', 'Rose', 'Zack'] 1888607607424  
['Alfred', 'Rose', 'Zack'] 1888607607424  
['Alfred', 'Rose', 'Zack'] 1888607607424  
['Alfred', 'Rose', 'Zack'] 1888607607424  
['Bernard', 'Suzan', 'Sasha'] 1888607161408
```

## Class Methods

We use DECORATORS to define *methods that are not based on instances* in classes.

We should use `@classmethod` right before defining the function, and instead of SELF, we use CLS as the argument so that it refers to the class, not the object.

Note that we can use any other name instead of SELF or CLS; these are well-known names to use, but not Mandatory system names. Only note that the first argument of in-class-method works as SELF (or CLS in case we have used the mentioned decorator before definition).

Class methods can be called like `className.methodName` without even a single instance of the class ever been made. But they're accessible

from objects too; for example, you can use *objectName.methodName* instead as well.

```
class site:
    users = ['Fernando', 'Goodwill', 'Leno']
    active = 0

    def __init__(self, name):
        if name in self.users:
            self.name = name
            print(f'{self.name} has logged in')
            site.active += 1
        else:
            print('this username has not been
registered')

    def logOut(self):
        print(f'{self.name} has logged out')
        site.active -= 1

    @classmethod
    def howmany(cls):
        print(cls.active)

site.howmany()
me = site('Goodwill')
you = site('Fernando')
he = site('Leno')
site.howmany()
he.logOut()
he.howmany()
```

```
0
Goodwill has logged in
Fernando has logged in
Leno has logged in
3
Leno has logged out
2
```

# Representation

if you type `print(objectName)`, it shows the **address** of the instance in memory. To change this action, we use function `__repr__`. See the difference:

<pre>class hello:     def __init__(self, name):         self.name = name  me = hello('Jona') print(me)</pre>	<pre>class hello:     def __init__(self, name):         self.name = name      def __repr__(self):         return f'hi {self.name}'  me = hello('Jona') print(me)</pre>
<pre>&lt;__main__.hello object at 0x000002662936B730&gt;</pre>	<pre>hi Jona</pre>

# Property

Imagine our class has an attribute and we don't want it to be shown or set or manipulated outside the class neglectfully; so, we use functions **getter** (to return the attribute) and **setter** (to control the value while being set). In this way we can have more control on our class's attrs.

Keep in mind that the attr we're talking about must have a different name than the function we define. To do so, we can have the *attr* and *func* names different by the capitalization of the first letter, or it's better to make the attr Private (in form `_attrName`). We use decorators like:

@property

def attrName():

commands

return

@attrName.setter

def attrName():

commands

```
class car:
    available = ['BMW', 'Porsche', 'Mazda']

    def __init__(self, br):
        self._brand = br

    # getter
    @property
    def brand(self):
        return self._brand

    # setter
    @brand.setter
    def brand(self, newBrand):
        if newBrand in self.available:
            self._brand = newBrand
        else:
            raise ValueError('this brand is ' \
                              'not available')
```

```
myCar = car('Ferrari')
print(myCar.brand)

myCar.brand = 'BMW'
print(myCar.brand)

myCar.brand = 'Benz'
print(myCar.brand)
print(myCar.brand)
```

Ferrari

BMW

Traceback (most recent call last):

File "D:\Python\Projects\venv\Test.py", line 27, in <module>

myCar.brand = 'Benz'

File "D:\Python\Projects\venv\Test.py", line 18, in brand

raise ValueError('this brand is not available')

ValueError: this brand is not available

Another example:

```
class fullName:
    def __init__(self, firstName, lastName):
        self.Name = firstName
        self.Family = lastName

    @property
    def name(self):
        return f'{self.Name} {self.Family}'

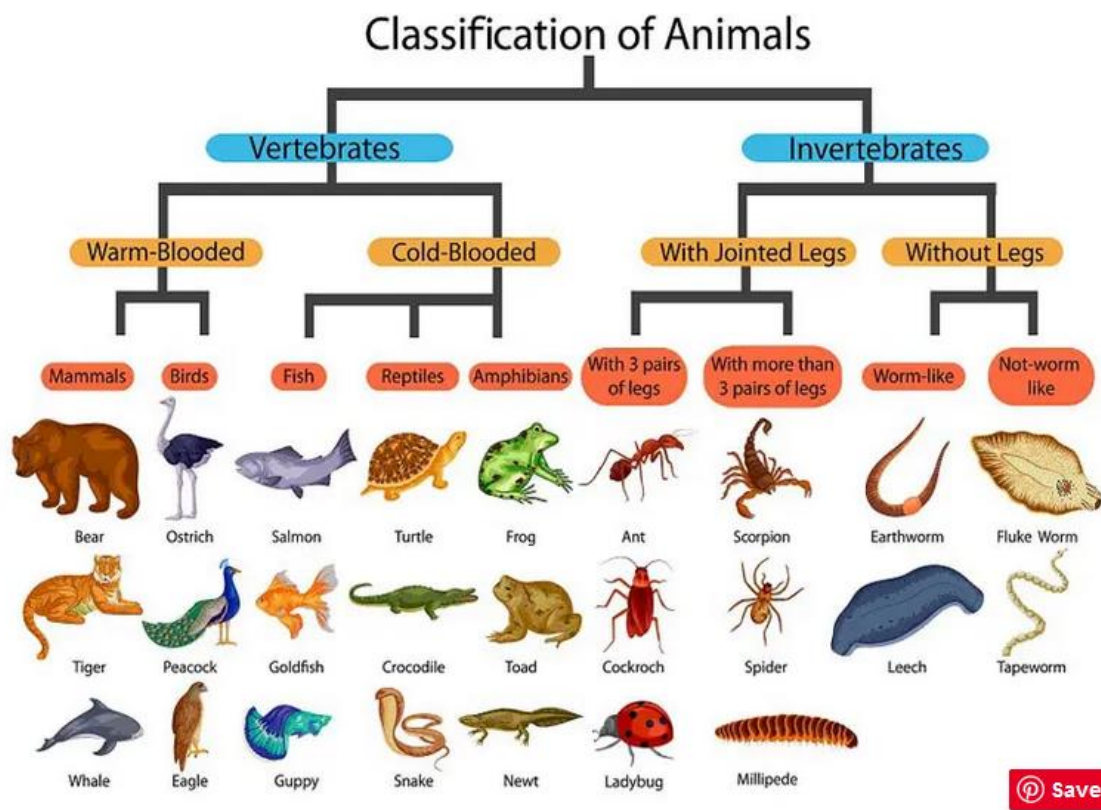
    @name.setter
    def name(self, full_name):
        n, f = full_name.split(',')
        if n != 'Amber':
            self.Name = n
        if f != 'Heard':
            self.Family = f
```

```
me = fullName('Jason', 'Momoa')
print(me.name)
me.name = 'Johnny Depp'
print(me.name)
```

Jason Momoa  
Johnny Depp

## Inheritance

Imagine you have a class, and there's another class which is a subset to that. This new class, has all the specifications of its parent class, and adds some more, specifically of its own. This concept is like animal category:



This concept works in Python via adding *parent class*'s name to the *subset class*'s while defining it:

```
class parent:
    commands
    commands

class subset(parent):
    commands
    commands
```

## Super

If the subset class has its own initial values that must be taken, the object must fulfill the initial requirements of the parent class in addition to those of subset class's. In the subset class, parent's parameters must be assigned using `super().__init__()` like:

```
class fullName:
    def __init__(self, firstName, lastName):
        self.name = firstName
        self.family = lastName
```



```

class user(fullName):
    def __init__(self, first, last, email):

        super().__init__(first, last)
        self.email = email

    def info(self):
        return f'{self.name} {self.family} is' \
               f' contacted via {self.email}'

me = user('Tom', 'Hanks', 'test@test.com')
print(me.info())

```

Tom Hanks is contacted via [test@test.com](mailto:test@test.com)

Instead of *super()*, we can use the parent class's name too; but in this case, we ought to put SELF as an argument too:

*parentClass.\_\_init\_\_(self, ..., ....)*

Note that if you don't use SUPER or last method for initiating parent class, the `__init__` method of PARENT won't be launched; so, when you are creating an instance of the SUBSET class, it will only ask for the subset requirements, not those of the parent class; although its funcs and attrs will still be still available. E.g.:

```

class par:
    def __init__(self, firstName, lastName):
        print('this is par')
        self.name = firstName
        self.family = lastName

    def hi(self):

```

```

        print(f'hi, par')

class sub1(par):
    def __init__(self, email, age):
        print('this is sub1')
        self.email = email
        self.age = age

    def info(self):
        return f'{self.age} years old is enough' \
               f' to have {self.email}'

class sub2(par):
    def __init__(self, n, f):
        super().__init__(n, f)
        print(f'mr {f}, welcome to sub2')

me = sub1('test@test.com', 20)
print(me.info())
me.hi()

you = sub2('Chris', 'Bale')

```

```

this is sub1
20 years old is enough to have test@test.com
hi, par
this is par
mr Bale, welcome to sub2

```

The last example is also a good practice to learn the order classes initiate based on. In conclusion, we learn that the `__init__` function of any parent class will be launched if we use a way to recall it. This will have more importance in *multiple* inheritance.

# Multiple Inheritance

When a class is subset to more than one parent class, multiple inheritance occurs, that we review its most important rules here, starting with an example:

```
class parent1:
    def __init__(self, firstName, lastName):
        self.name = firstName
        self.family = lastName

    def hi(self):
        print(f'hi {self.name}, 1')

class parent2:
    def __init__(self, email):
        self.email = email

    def info(self):
        return f'{self.name} {self.family} has ' \
               f'{self.age} years old and can be ' \
               f'contacted with via {self.email}'

    def hi(self):
        print(f'hi {self.name}, 2')

class subset1(parent1, parent2):
    def __init__(self, fir, las, em, age):
        super().__init__(fir, las)
        # same as : parent1.__init__(self, fir, las)

        parent2.__init__(self, em)
        self.age = age

class subset2(parent2, parent1):
    def __init__(self, fir, las, em, age):
        super().__init__(em)
```

```

        # same as : parent2.__init__(self, em)

    parent1.__init__(self, fir, las)
    self.age = age

Hulk = subset1('Mark', 'Ruffalo', 'test@test.com', 54)
print(Hulk.info())
Hulk.hi()

Venom = subset2('Tom', 'Hardy', 'test@test.com', 44)
print(Venom.info())
Venom.hi()

```

Mark Ruffalo has 54 years old and can be contacted with via test@test.com  
 hi Mark, 1  
 Tom Hardy has 44 years old and can be contacted with via test@test.com  
 hi Tom, 2

In this example, both *subset1* and *subset2* inherit from 2 parent classes above them. Note that we list parent classes in parentheses in front of subset class's name while defining it: *subset(parent1, parent2, ...)*

Another important fact in this example is the order in which we mention parent classes in these parentheses. As you see in the results, for *Venom* which is an instance of *subset2*, calling the *hi()* func executes the one in *parent2*; that's because we mentioned *parent2* as the first parent class while defining *subset2*.

As a matter of fact, if we have funcs or attrs with the same name in parent classes, and we want to use them, the one that will be launched is the one belonging to the first parent class we've mentioned in definition of the subset class. The same order applies for initiating the *\_\_init\_\_* func of parent classes by using *super().\_\_init\_\_(.....)*.

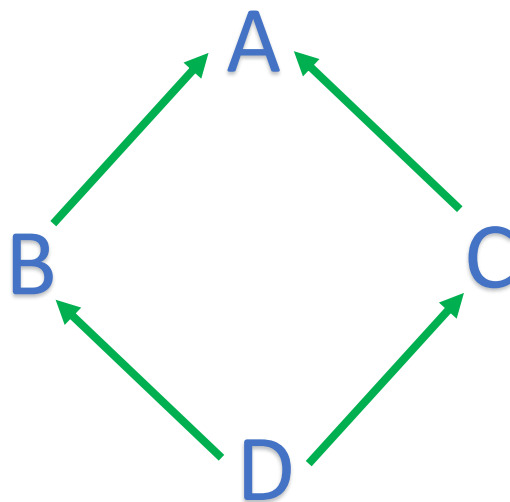
You see another lesson if you take a look at *info* func in *parent2*; although there is no *age* defined in this class, the code works! That is because the SELF refers to the INSTANCE, which is an object of SUBSET class, so all attrs of this class AND its parents are available for the instance and SELF.

*isinstance(objectName, className)* → shows if an object is the instance of a particular class or not.

Here emerges the importance of name mangling; if we have attrs with the same name in different classes, how do we professionally distinguish them? Of course, by name mangling.

## Method Resolution Order

Imagine we have some classes with the relationship in below, and all of them have a func with the same name; so, when we call THE function, which one do you think will be launched?



There's a built-in method to let us find out the sequence in which these class functions are executed. We can use either of these 3 ways:

- `.__mro__`
- `.mro()`
- `help()`

The fact is, when we have a func with the same name in all these classes, while being called, it searches the main class of which our object is an instance of, first. Then, if not found, it searches the closest parent classes, with the order they've been introduced, and if not available in any of them, goes to the grand-parent(!)

```
class A:
    def hi(self):
        print('hi A')

class B(A):
    def hi(self):
        print('hi B')

class C(A):
    def hi(self):
        print('hi C')

class D(B, C):
    def hi(self):
        print('hi D')
```

```
item = D()
item.hi()

print(D.__mro__)
print(D.mro())
print(help(D))
```

hi D

(<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>)

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

Help on class D in module \_\_main\_\_:

class D(B, C)

| Method resolution order:

| D  
| B  
| C  
| A  
| builtins.object

| Methods defined here:

| hi(self)

| -----  
| Data descriptors inherited from A:

| \_\_dict\_\_  
| dictionary for instance variables (if defined)  
|  
| \_\_weakref\_\_  
| list of weak references to the object (if defined)

None

# Polymorphism

There is a widely accepted contract that if multiple classes have some similar funcs because they are `_somehow_` members of the same group, we make these classes subset of a parent class which has the mentioned common funcs. Yet we still define the functions in every class as well. For example, this is a common way of such situations:

```
class Ianimal: # the 'I' at the first
               # stands for 'Interface';
               # which is the common name
               # this kind of classes
               # are called by

    def sound(self):
        raise NotImplementedError

class dog(Ianimal):
    def sound(self):
        print('wagh wagh')

class cat(Ianimal):
    def sound(self):
        print('miu miu')

pet1 = dog()
pet1.sound()

pet2 = cat()
pet2.sound()
```

```
wagh wagh
miu miu
```



# Dunder Methods

There are some built in functions or operators like `+` or `len()` that act differently for different classes. For example, `2 + 2` returns `4`, while `'2' + '2'` returns `'22'`. (Note that **types** *like* `int`, `float`, `str`, `list`, `etc.` are actually **classes**)

We can choose what this kind of methods must do in our class. Follow the example:

```
class info:
    def __init__(self, name, family, age, money):
        self.nam = name
        self.fam = family
        self.age = age
        self.mon = money

    def __repr__(self):
        return f'{self.nam} {self.fam} is {self.age}' \
            f' years old & owns {self.mon}$'

    def __add__(self, other):
        return self.age + other.age

    def __sub__(self, other):
        res = self.mon - other.mon
        if res >= 0:
            return res
        else:
            return f'{self.nam} has less money' \
                f' than {other.nam}!'

    def __mul__(self, other):
        return f'{self.nam} {other.fam}'

    def __truediv__(self, other):
        return self.mon / other.mon

    def __len__(self):
        return self.age
```

```

def __copy__(self):
    return self

def __sizeof__(self):
    return len(self.nam)

Peter = info('Andrew', 'Garfield', 38, 20000)
Parker = info('Tobey', 'Maguire', 47, 12000)

print(Peter + Parker) # 85
print(info.__add__(Peter, Parker)) # 85

print(Parker - Peter) # Tobey has less money than Andrew!
print(info.__sub__(Peter, Parker)) # 8000

print(Peter * Parker) # Andrew Maguire
print(Peter / Parker) # 1.6666666666666667
print(len(Peter), len(Parker)) # 38 47
print(Peter.__sizeof__(), Parker.__sizeof__()) # 6 5

me = Peter.__copy__()
print(me) # Andrew Garfield is 38 years old & owns 20000$

you = info.__copy__(Parker)
print(you) # Tobey Maguire is 47 years old & owns 12000$

```

```

85
85
Tobey has less money than Andrew!
8000
Andrew Maguire
1.6666666666666667
38 47
6 5
Andrew Garfield is 38 years old & owns 20000$
Tobey Maguire is 47 years old & owns 12000$

```

# Iterator

We have introduced **iterables** very simply before. We assumed every variable with multiple indexes as iterables. In a more precise definition, iterable is called to **any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop**. Familiar examples of iterables include lists, tuples, and strings - any such sequence can be iterated over in a for-loop.

**Iterators** \_despite their small difference\_ are usually mistaken for iterables (i.e. they are called to the same thing as iterables.) In fact, Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from. All these objects have a *iter()* method which is used to get an iterator.

In other words, an iterator is an object that contains a countable number of values (i.e. an object that can be iterated upon, meaning that you can traverse through all the values.)

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

As an example of how an iterator is **returned** from an iterable, and how the two methods mentioned above, work, see the following code:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(mytuple))
```

```
apple
banana
cherry
print(next(mytuple))
TypeError: 'tuple' object is not an iterator
```

To learn better how **for** loops do iteration, we stimulate them like:  
(note: Another important error type any programmer might face with is **StopIteration** err; learn about it more in the comments)

```
nums = [1, 2, 3]

for num in nums:
    print(num * 2)

print('-----')

numbers = iter(nums)
print(2 * next(numbers)) # 1
print(2 * next(numbers)) # 2
print(2 * next(numbers)) # 3
# print(2 * next(numbers)) # 4 =>
# this line causes an StopIteration error,
# so we handle this job better with a WHILE loop:

print('-----')

numbers_2 = iter(nums)
while True:
    try:
        print(2 * next(numbers_2))
    except StopIteration:
        break
```

2	2	2
4	4	4
6	6	6
-----	-----	

To create an object as our own iterator, we have to implement the methods `__iter__()` and `__next__()` to your object.

In the `__iter__()` method, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

As an example, we simulate `RANGE()` method like this:

```
class ranger:
    def __init__(self, stop, start=1, step=1):
        self.current = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        return self

    def __next__(self):
        if self.step > 0:
            if self.current < self.stop:
                n = self.current
                self.current += self.step
                return n
            raise StopIteration
        elif self.step < 0:
            if self.current > self.stop:
                n = self.current
                self.current += self.step
                return n
            raise StopIteration

counter = ranger(25, 5, 5)
# remember: stop, start, step

for num in counter:
    print(num)
```

```
print('-----')

counter = ranger(5, 25, -5)
for num in counter:
    print(num)

print('-----')

counter = ranger(6)
for num in counter:
    print(num)
```

```
5
10
15
20
-----
25
20
15
10
-----
1
2
3
4
5
```

# Generator

Generator Is a kind of iterators. We use generator functions mostly to use them as an iterable. Generators have **yield** instead of RETURN, so they have some differences. The best advantage of a generator func rather than simple ones is, generator funcs don't load all the data at the same time; so, for HUGE operations, they use little memory, while simple funcs can't even handle them. Another difference is, in generator funcs, the code after *yield* is executed, but in simple ones, what you add after *return* doesn't count. Besides, generator funcs preserve their last state, as you see:

```
def ranger(stop):
    count = 1
    while count <= stop:
        yield count
        count += 1

print(ranger(5))
counter = ranger(5)
print(next(counter)) # 1
print(next(counter)) # 2
print(next(counter)) # 3
print(next(counter)) # 4
print(next(counter)) # 5
print(next(counter)) # ERR
```

<generator object ranger at 0x000001811A91ABA0>

1  
2  
3  
4  
5

Traceback (most recent call last):

print(next(counter)) # ERR

StopIteration

Actually, in the last example, what happens is: when the func is being called, it runs up to the *yield* line and stops; the next time it's being called by the *next()* command, it resumes from where it'd stopped and continues the loop. For example, if we use *yield* instead of *return* in the code below, the performance would be uninterrupted even for a million items of Fibonacci list, while using *return*, it will most likely run short of RAM in most systems.

```
def fib(items): # 1,1,2,3,5,8,13,21,34,...
    res = [1]
    a, b = 0, 1
    count = 2
    while count <= items:
        a, b = b, a + b
        res.append(b)
        count += 1
    return res

for f in fib(1000000):
    print(f)
```

We can use **comprehension** method for generators as well in this way:

```
def num1():
    for n in range(1, 10):
        yield n

numbers = num1()
print(next(numbers))
print(next(numbers))
print(next(numbers))

print('----')
```



```
num2 = (n for n in range(1, 10))
print(next(num2))
print(next(num2))
print(next(num2))

print('----')

print(sum((n for n in range(100))))
# works for very huge numbers

print(sum([n for n in range(100)]))
# while this one, crashes
```

```
1
2
3
----
1
2
3
----
4950
4950
```

# Decorator

Decorator is just a function that gets another func as its parameter, and returns another func which is inside it. In fact, decorators are used for modification or correction of other funcs. E.g.:

```
# be careful about the usage of parentheses
# whenever a function name is mentioned

def intro(func):
    def hi_bye():
        print('Hello')
        func()

    return hi_bye

def outro():
    print('GoodBye')

test = intro(outro)
test()
```

Hello  
GoodBye

This code should be rewritten for a more professional form in this way:

```
def intro(func):
    def hi_bye():
        print('Hello')
        func()

    return hi_bye
```

```
@intro
def outro():
    print('GoodBye')

test = outro()
```

Hello  
GoodBye

In a more advanced case, when the func has parameters:

```
def intro(func):
    def hi_bye(_name):
        # this function MUST have a parameter like OUTRO has

        print(f'Hello {_name}')
        func(_name)
        # this function MUST have a parameter too
        # because it's originated from OUTRO

    return hi_bye

@intro
def outro(name):
    print(f'Goodbye {name}')

test = outro('Kiano')
```

Hello Kiano  
Goodbye Kiano

(we'll describe the more complicated case where both *decorator* and the *func* have their own parameters later)

A more practical and sensible example:

```
def title(func):
    sexuality = 'man'

    def res(_name, _score):
        if sexuality == 'man':
            print('Hello Mr.', end=' ')
        elif sexuality == 'woman':
            print('Hello Ms.', end=' ')

        func(_name, _score)

    return res

@title
def message(name, score):
    if score >= 50:
        print(f'{name}, you\'ve passed the exam.')
    else:
        print(f'{name}, you\'ve failed the exam.')

test = message('Liam', 67)
```

Hello Mr. Liam, you've passed the exam.

Since different functions might use our decorator, we can't fix a limit for parameters of inner function of our decorator; therefore, we use \*args. E.g.:

```
def courtesy(func):
    def greeting(*arg):
        func(*arg)
        print(f'it\'s a pleasure to have you!',
              '\n-----')

    return greeting
```

```

@courtesy
def intro(name, family):
    print(f'Hello {name} {family}.')

@courtesy
def outro(name):
    print(f'GoodBye {name}.')

@courtesy
def login():
    print('welcome to our site!')

test1 = intro('Samuel', 'L. Jackson')
test2 = outro('Bruce')
test3 = login()

```

```

Hello Samuel L. Jackson.
it's a pleasure to have you!
-----
GoodBye Bruce.
it's a pleasure to have you!
-----
welcome to our site!
it's a pleasure to have you!
-----

```

We use a special decorator called `wraps()` from *functools* module on the wrapper function of our decorator, so that the functions on which our decorator is set, preserve their identity. Look what happens if we don't: (don't worry about *modules*, they are just pre-written useful codes ready for us to use; we will reach them soon enough :))

```

def Hello(func):
    def wrapper():
        print('Hello, it\'s a pleasure to meet you!')
        print(func.__name__)
        # fortunately, it returns the true name
        # of the function passed to this decorator
        func()

    return wrapper

@Hello
def Goodbye():
    print('Goodbye sir. hope to see you again!')

test = Goodbye()
print(Goodbye.__name__)
# unfortunately, it shows the name of the
# decorator func, instead of the functions true name

```

Hello, it's a pleasure to meet you!  
 Goodbye  
 Goodbye sir. hope to see you again!  
 wrapper

On the other hand:

```

from functools import wraps
# we import modules like this
# you'll learn about it later

def Hello(func):
    @wraps(func)
    def wrapper():
        print('Hello, it\'s a pleasure to meet you!')
        print(func.__name__)
        func()

    return wrapper

```

```
@Hello
def Goodbye():
    print('Goodbye sir. hope to see you again!')

test = Goodbye()
print(Goodbye.__name__)
```

Hello, it's a pleasure to meet you!  
 Goodbye  
 Goodbye sir. hope to see you again!  
 Goodbye

## Decorator Factory

We've reviewed the case which the function decorator works upon gets some parameters. But what if the decorator itself has parameters of its own? If we want to send some data to our decorator independently, we put our entire decorator in another function called decorator factory, which is responsible of receiving THE data. E.g.:

```
from functools import wraps

def dec_fac(maxPrice):
    def inner_func(func):
        @wraps(func)
        def wrapper(name, cost):
            func(name, cost)
            if cost > maxPrice:
                print('too expensive to afford!')
            else:
                print('purchase succeeded!')

        return wrapper

    return inner_func
```

```
@dec_fac(1000)
def cart(item, price):
    print(f'{item} selected;')

test = cart('Lexus LX570', 2000)
test = cart('Lexus RX350', 999)
```

Lexus LX570 selected;  
too expensive to afford!  
Lexus RX350 selected;  
purchase succeeded!



# File

To open a file, use `var = open('file address')`. This method saves the file's content in the var. you can add more options such as *mode* too. Mode *'r'* is for reading, *'w'* for rewriting from the first, *'a'* to append to the file, *'r+'* for reading and writing, and, finally, *'a+'* for reading and appending.

Use `var.write(str_to_be_written)` for both writing or appending.

While opening the file, for mentioning its address, its sufficient to use only files name like `'file_name'` if the file is in the directory folder of your Python file.

To print what you've saved, use `print(var.read())`. In this method, if you add a number as an argument of `.read()`, it reads as many indexes as you desire.

This last method reads the file up to its last index; and the curser remains there; so, the next time you call this method, it doesn't show anything.

To change the curser position, use `var.seek(index_number)`. Use zero to go to the beginning of file.

The method `var.readline()` reads text line by line. And `var.readlines()` saves the text as a list of its lines.

Use `var.close()` to close what you've opened, in order to reduce the memory and energy consumption of system.

If you use the syntax below (working with file in a paragraph), it will close the file automatically after paragraph ends.

```
with open('test.txt') as myFile:
    A = myFile.read(14)
    print(A)
```

In the name of

```

execution = open('C:/Users/Mohammad/Desktop/test.txt')
print(execution.read())    # reads entire file

a = execution.readline()   # reads nothing because the
print(a)                   # curser is currently at the
                           # end of the file

execution.seek(0)          # moves the curser to the
                           # start of the file

a = execution.readline()   # reads the first line of the
print(a)                   # file

b = execution.readlines()  # reads the rest of the file
print(b)                   # (from line 2) putting every
                           # line as a segment of list 'b'

execution.close()

```

In the name of the king  
 Robert of the house Baratheon  
 the first of his name  
 the king of the Andals and the First men  
 the lord of seven kingdoms  
 and protector of the realm

I  
 Eddard of the house Stark  
 lord of Winterfell  
 and warren of the north  
 do hereby  
 denounce you for treason  
 and sentence you to death  
 may the gods have mercy upon your soul!

In the name of the king

['Robert of the house Bratheon\n', 'the first of his name\n', 'the king of the Andals and the First  
 men\n', 'the lord of seven kingdoms\n', 'and protector of the realm\n', '\n', '\n', 'edward of the  
 house stark\n', 'lord of winterfel\n', 'and warren of the north\n', 'do hereby\n', 'denounce you  
 for treason\n', 'and sentence you to death\n', 'may the gods have mercy upon your soul!\n']

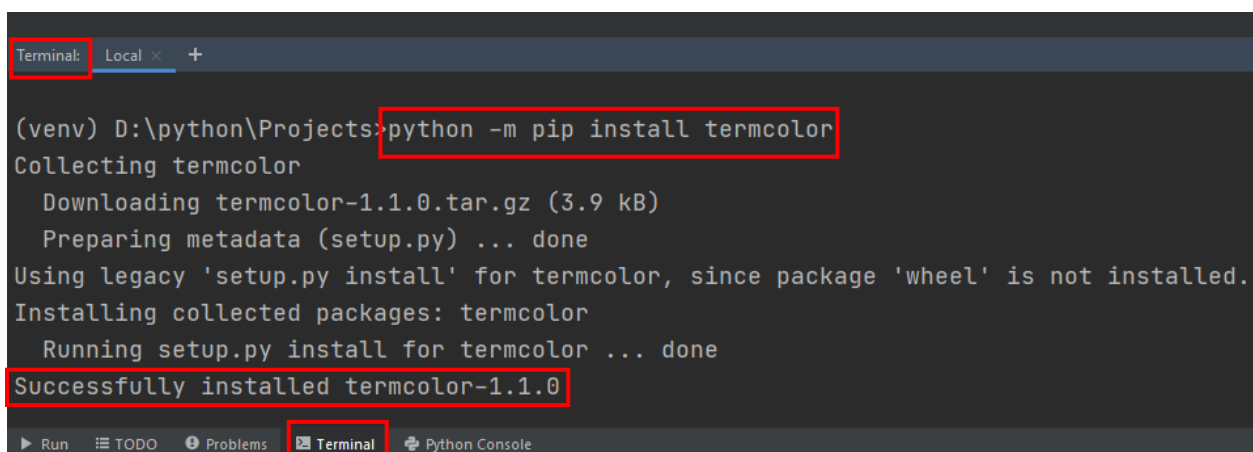
# Modules

Modules, also known as packages or libraries, are pre-written codes some individuals or companies had published as a tool or aid for programmers.

Actually, modules have special methods, classes, constants, etc. of specific areas that any developer can import them and use them easily instead of programming them by oneself.

Some basic modules (called standard packages) are pre-installed (i.e., come with the Python interpreter) so you can import them straightly whenever required. Others, in contrast, must be first installed on the system so the compiler can find their source code.

To install a non-standard package, we use Python's package manager called **pip**. We head to the Terminal or CMD and type ***Python -m pip install package name***. If the system is connected to the internet, it will automatically download and install it and show success message. ***pip uninstall package name*** uninstalls unwanted packages.



```
Terminal: Local x +
(venv) D:\python\Projects>python -m pip install termcolor
Collecting termcolor
  Downloading termcolor-1.1.0.tar.gz (3.9 kB)
  Preparing metadata (setup.py) ... done
Using legacy 'setup.py install' for termcolor, since package 'wheel' is not installed.
Installing collected packages: termcolor
  Running setup.py install for termcolor ... done
Successfully installed termcolor-1.1.0
```

To see the list of all modules installed on your system, type **Python** at terminal; then, using the command **help('modules')** you can find your list. Or just use **pip list** method easily. To see if a particular package exists on your system, use **help('modules packageName')**.

```
(venv) D:\python\Projects>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help('modules')

Please wait a moment while I gather a list of all available modules...
```

If a package is available for the compiler, we can finally import and use it. There are multiple ways to import a package:

- **import ..... (as .....)**
- **from ..... import .....**
- **from ..... import ..... (as .....), ..... (as.....)**
- **from ..... import \***

Using the first way, we have to address the package name whenever we use its contents in our code. The **as .....** part is optional while importing. What it does is, for long package names, it lets you add an abbreviation to address the package with, instead of its original name. This helps cleanliness and optimization of the code, since sometimes we ought to import something with very long name; so instead of this name which takes too much space, we use only a couple of letters of our own, to abbreviate it. Some famous examples are:

```
import matplotlib.pyplot as plt
import numpy as np
import tkinter
```

In the second way, we only import a particular content of the package.

The third method, is an advanced form of the last one; only here, we import more than one thing, and for each one, we can add an abbreviation too.

With the fourth one, we import all package's contents, but we don't have to address them with any kind of package name when we want to use them.

To see the guide or catalogue of a package and its contents, use [`help\(packageName\)`](#) after importing the package.

Review all these with the example below:

```
import matplotlib.pyplot as plt
from numpy import pi
from math import sin, cos, factorial as fac
from tkinter import *

# examples of addresseing the modules:
# plt.grid()          -> matplotlib.pyplot
# print(pi)           -> numpy
# cos(12) / fac(4)    -> math
# a = Label()         -> tkinter

help(plt)
```

Help on module matplotlib.pyplot in matplotlib:

NAME

matplotlib.pyplot

DESCRIPTION

`matplotlib.pyplot` is a state-based interface to matplotlib. It provides an implicit, MATLAB-like, way of plotting. It also opens figures on your screen, and acts as the figure GUI manager.

.  
.
.

# Numpy

Numpy is a very useful module for numerical and mathematical calculations. Numpy works mostly with matrixes and can be recalled as Matlab's equivalent in Python.

## MATRIX

To make a matrix out of a list, use `array(listName)`. Then, you're able to do mathematical operations with matrices:

```
import numpy as np

a = [1, 2, 3]
b = [2, 4, 6]
# a 2D matrix:
c = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

A = np.array(a)
B = np.array(b)
C = np.array(c)

print(A)
print(C)
print(a + b)      # mind the difference
print(A + B)      # of these two lines
```

```
[1 2 3]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[1, 2, 3, 2, 4, 6]
[3 6 9]
```

Address matrix indexes like lists. For multi-dimensional ones, however, you can both use `var[...][...]` or merge two brackets like `var[...,...]`. As for lists, here, too, you can use `:` to choose all columns or all rows. For instance, following previous example:

```
print(A[1])
print(C[0])
print(C[0][1])
print(C[0,1])
```

```
2
[1 2 3]
2
2
```

## USEFUL METHODS

`linlag.norm()` → to calculate the norm of a vector.

(vector = [x, y, z] → vector's norm =  $\sqrt{x^2 + y^2 + z^2}$  )

`arange()` → works like standard `range()` method, except it can get float arguments, while `range()` only works with integers.

`ones((a,b))` → creates a unit matrix  $a \times b$

`zeros((a,b))` → creates a null (zero) matrix  $a \times b$

`shape` → returns matrix's size (in matter of dimentions)

`cross(a,b)` → returns cross multiplication of matrices a and b.

Using `A*B` multiplies every segment of 'A' matrix, one by one, to the correspondent segment on 'B' matrix.

To multiply two matrices with the matrix multiplication, use `A@B`.

`random()` → provides lots of tools for *random* choices. (Another independent module for the same purpose is '*random*'<sup>1</sup>)

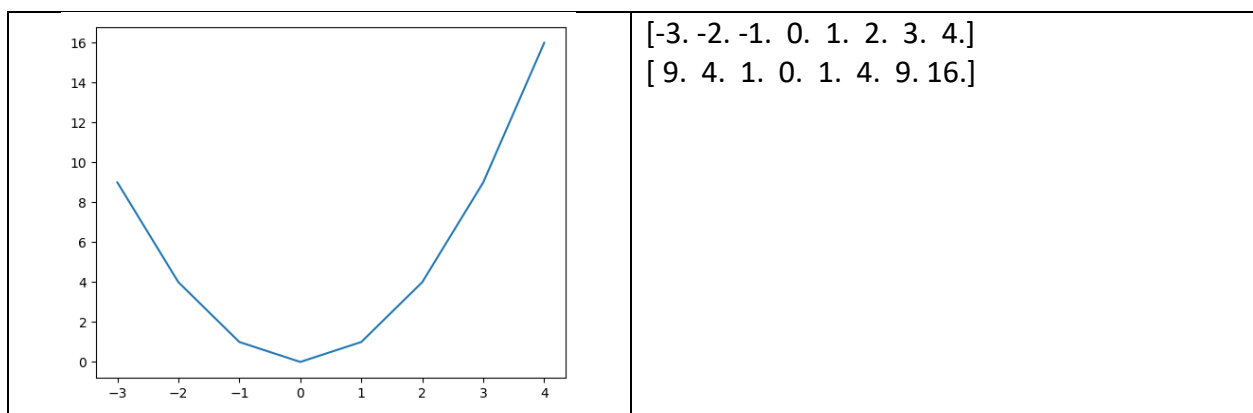
## FILE

There is an easy way to read text files in numpy; all you've got to do, is to use `numpy.loadtxt('file name', delimiter = 'the char(s) that are used to separate data')`. See the following example: (you can compare it with the method taught in the CSV module which you'll learn about later on)

```
import numpy as np
import matplotlib.pyplot as plt
# you will learn about this last module later,
# it's used here only to draw the data's plot.

x, y = np.loadtxt('test.txt', delimiter=' | ',
                  unpack=True, skiprows=1)

print(x)
print(y)
plt.plot(x, y)
plt.show()
```



<sup>1</sup> [https://www.w3schools.com/PYTHON/module\\_random.asp](https://www.w3schools.com/PYTHON/module_random.asp)



test.txt:

```
DATA :  
-3 | 9  
-2 | 4  
-1 | 1  
0 | 0  
1 | 1  
2 | 4  
3 | 9  
4 | 16
```

As you see, the txt file here, has a header before its numerical data. The `loadtxt()` method, however, can't deal with strings, but only with numbers. Therefore, we added `**kw skiprows` to skip the headline and prevent causing the following err:

**ValueError: could not convert string to float: 'DATA:'**



# Sympy

This module is used for mathematical and scientific uses, especially symbolic calculations, such as integration, differentiation, solving an equation, value substitution and etc.

## DEFINITION OF SYMBOLIC VARIABLES

There are two for this matter:

```
import sympy as sp

x, y, z = sp.symbols('x,y,z') # or ('x y z')
t = sp.Symbol('t')
# note the uppercase letters
```

## SUBSTITUTION OF VALUES

If you want to substitute a sym var with a value or anything else:

```
x, y = sp.symbols('x,y')
z = 2 * x + y

print(z.subs(x, 2))      # subs a numeric value
print(z.subs(x, y))      # subs a symbolic var
t = z.subs(x, y)         # save data in another var
print(z)                 # main expression doesn't change
print(t)
print(z.subs(x, 2).subs(y, 4)) # concatenate subs
```

```
y + 4
3*y
2*x + y
3*y
8
```

## SHOWING THE RESULTS

Use `.evalf()` to get a numerical version of an expression, as a *float*. to format a mathematical expr in a neat way, use `pretty_print()`: (this might be known as sympy's equivalent of `pretty()` function in Matlab.)

```
x, y = sp.symbols('x, y')  
  
a = sp.pi + sp.exp(1)  
b = 2 * x + sp.sqrt(x ** 2) / sp.asin(y)  
print(a)  
print(a.evalf())  
print(b)  
sp.pretty_print(b)
```

E + pi

5.85987448204884

2\*x + sqrt(x\*\*2)/asin(y)

$$2 \cdot x + \frac{\sqrt{x^2}}{\sin(y)}$$

## SYSTEM OF EQUATIONS

As an equivalent of Matlab's `solve()` function, we solve system of equations with such policy:

```
x, y, z = sp.symbols('x, y, z')  
  
a = sp.Eq(-2 * x + y, z) # -2 * x + y = -z  
b = sp.Eq(-x + 2 * y + z, y)  
c = sp.Eq(2 * x ** 2 + y + z, 5 * x)
```

```
# or:
# a = -2 * x + y - z
# b = -x + 2 * y + z - y
# c = 2 * x ** 2 + y + z - 5 * x

res = sp.solve([a, b, c], (x, y, z))
# x= 2, y = 3, z = -1
print(res)
```

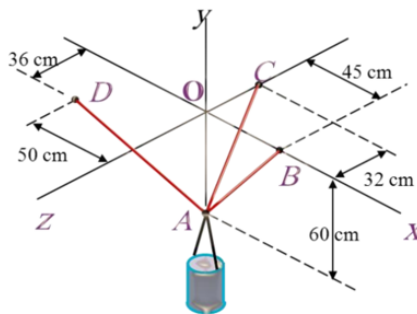
[(0, 0, 0), (2, 3, -1)]

As another example, see the solution for this STATIC problem:

(Only suggested to engineers who have passed the Static course)

## Problem

$F(A)=1165\text{N}$   
AB,AC,AD tensions=?



Solution:

```
from numpy import *
import numpy.linalg as nl
import sympy as sp

ab = array([45, 60, 0])
ac = array([0, 60, -32])
ad = array([-50, 60, 36])
```

```

b, c, d = sp.symbols('b,c,d')

ab = ab / nl.norm(ab)
ac = ac / nl.norm(ac)
ad = ad / nl.norm(ad)

AB = ab * b
AC = ac * c
AD = ad * d

q1 = AB[0] + AC[0] + AD[0]
q2 = AB[1] + AC[1] + AD[1] - 1165
q3 = AB[2] + AC[2] + AD[2]

s = sp.solve((q1, q2, q3), (b, c, d))
print(s)

```

{b: 500.0000000000000, c: 459.0000000000000, d: 516.0000000000000}

Check out [Solvers - SymPy 1.11 documentation](#)<sup>1</sup> for further info.

## DIFFERENTIATION

Use `.diff(expr, var)` to differentiate the *expr* based on the differentiation factor *var*. E.g.:

```

x = sp.Symbol('x')
y = x ** 2 + sp.cos(x)
yprime = y.diff(x)
print(yprime)

```

2\*x - sin(x)

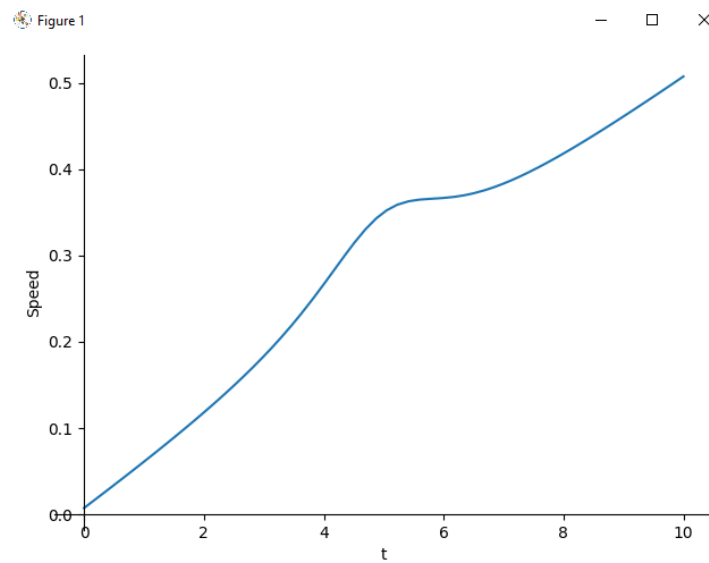
---

<sup>1</sup> <https://docs.sympy.org/latest/modules/solvers/solvers.html>

# PLOT

To draw the diagram of a symbolic expr, use `plot()`:

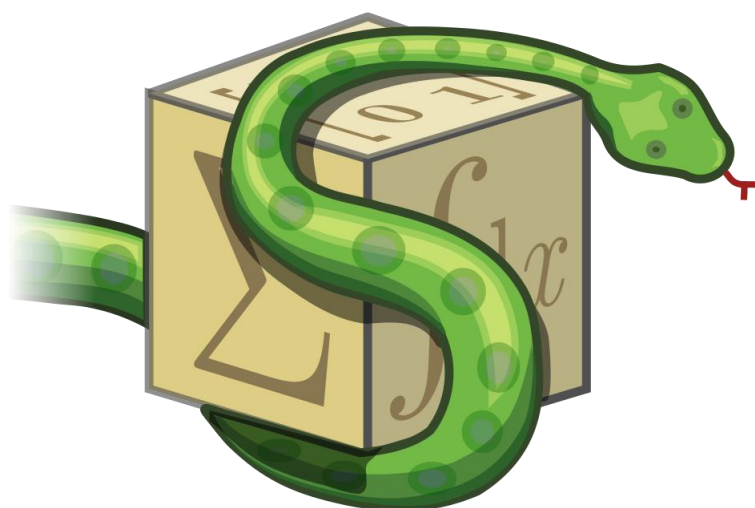
```
t = sp.symbols('t')
x = 0.05 * t + 0.2 / ((t - 5) ** 2 + 2)
sp.plot(x, (t, 0, 10), ylabel='Speed')
# or use:
# sp.plotting.plot(x)
```



Check out [Plotting - SymPy 1.11 documentation](https://docs.sympy.org/latest/modules/plotting.html)<sup>1</sup> for further info.

---

<sup>1</sup> <https://docs.sympy.org/latest/modules/plotting.html>



# SymPy

## Scipy

Scipy is a package similar to sympy, but more *numerical* AND *scientific* rather than *symbolic*. For example, integration method in this package only supports definite integrals and returns a numeric answer only.

There are tons of tools available in this package most useful for engineering stuff. Check out [SciPy](https://scipy.org/)<sup>1</sup> documentation for further info.



---

<sup>1</sup> <https://scipy.org/>

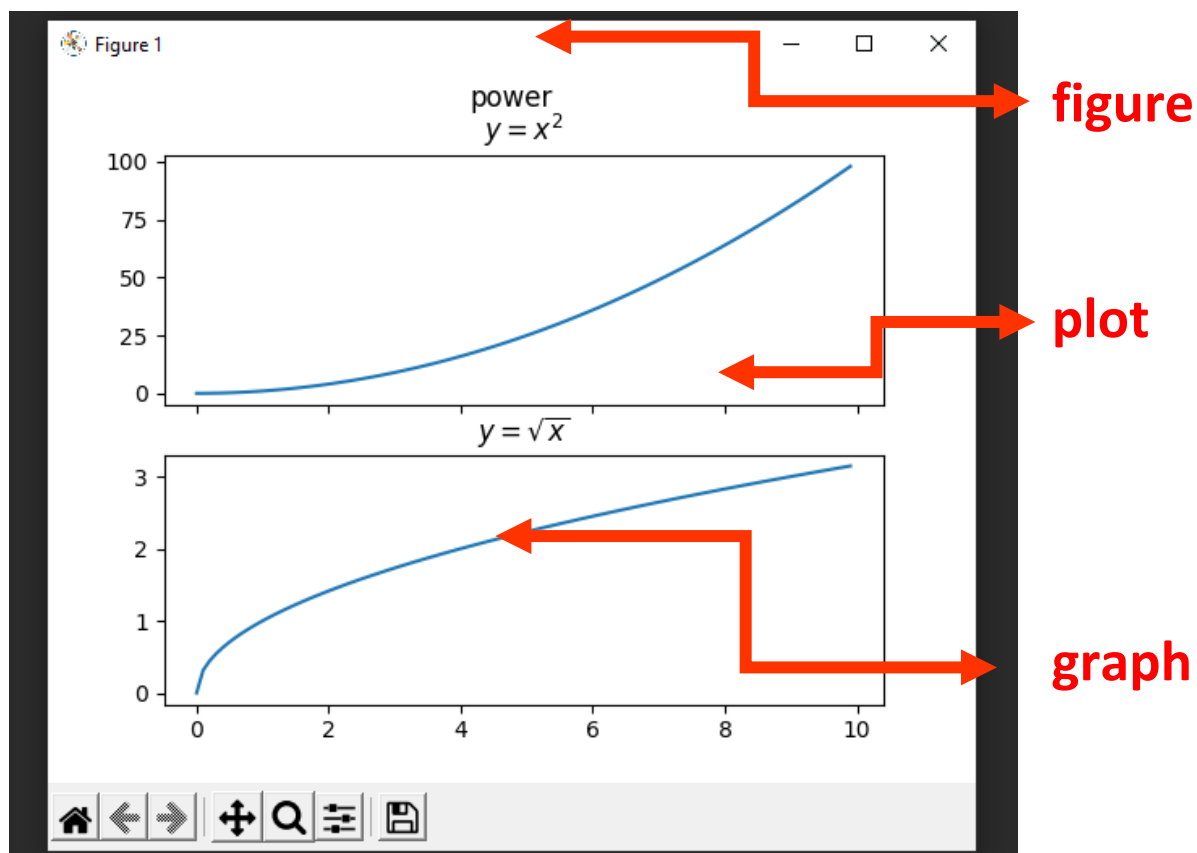


# Matplotlib

One of the most useful and well-known modules of Python, is this one. Matplotlib, with its huge variety of functions, is one of the best tools to demonstrate data as diagrams.

Most of the time, it's a special part of this enormous module that we will utilize, and that's called *pyplot*. We'll learn plotting methods from simple to more complex cases gradually.

Three important concepts of this module we deal with in this notebook, are *figure*, *plot* and *graph*. It's important to be able to distinguish them apart. In fact, a *figure*, is the window in which your *plots* are. These *plots*, themselves, can have multiple *graphs* inside. There also can be more than one figure to be shown with your code, and even more than one plot in each one.



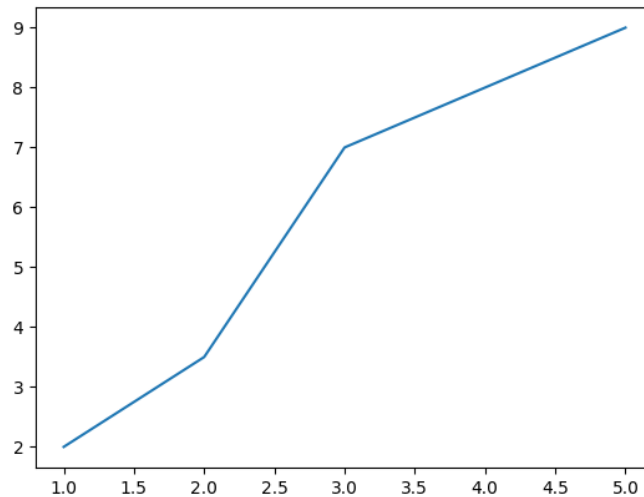
# PLOT

Using `plot(x, y)` draws the plot of `y`, based on `x`. Both arguments are iterables. You can also add more iterables and they will all be drawn based on `x` as well. Use `show()` in the end to execute the results:

```
import matplotlib.pyplot as plt

x = range(1, 6)
y = [2, 3.5, 7, 8, 9]

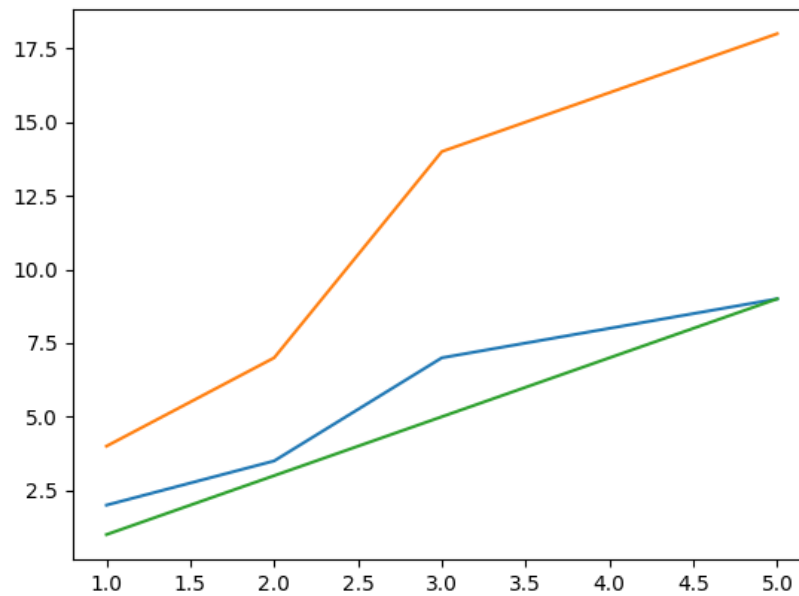
plt.plot(x, y)
plt.show()
```



You can also use `plot(...)` multiple times; in this case, use `show()` func only once, in the end, to draw all the graphs in one plot. E.g.:

```
x = range(1, 6)
y = [2, 3.5, 7, 8, 9]
z = [2 * n for n in y]
t = range(1, 11, 2)

plt.plot(x, y)
plt.plot(x, z)
plt.plot(x, t)
plt.show()
```



## Useful Customization Methods:

`plt.grid()` → adds grid to your plot.

`plt.ylabel('.....')` → adds a name for Y axis.

`plt.xlabel('.....')` → adds a name for X axis.

Use `plt.title("title_name")` for titling your plot. Use `set_title()` instead of `title()` if you're showing more than one plot in your figure.

`plt.suptitle('figure title')` and `plt.gcf().canvas.set_window_title('window title')` and `plt.figure('window title')` are more advanced methods.

There are dozens of pre-figured styles in Matplotlib. To see a complete list, use the `print(plt.style.available)` and to apply one of these styles, use `plt.style.use('styleName')`.<sup>1</sup>

---

<sup>1</sup> See the full list of these styles in appendix

To save a figure, you can use `plt.savefig('a name you've chosen your plot to be saved as')`. With this, your figure will be saved to the directory in which your code is; to choose another place, add its path before the mentioned name.

Use `plt.text(x, y, 'text')` to add a text on your plot. There's so much customization for texture and annotation. Have a glance at matplotlib's official website.<sup>1, 2, 3</sup>

Use `plt.axis('equal')` or `plt.axis('square')` to have your plot's both axes in the same scale.

Use `plt.xlim(lower-bound, upper-bound)` to restrict the range shown on x-axis; same method for y axis is `plt.ylim(..., ...)`.

Use `plt.axis([xmin, xmax, ymin, ymax])` to bound both plot axes to specific values at the same time.<sup>4</sup>

Use `plt.legend()` to add a guide about what's in your plot. To do so, you need to label everything you're showing on your plot while defining it. You can change legend's location with `loc` kwarg, like:

`plt.legend(loc='upper-left')`.

If you don't want to declare each graph's label in its definition line, you can give plots' label to `legend()` itself, like: `plt.legend(['first graph', 'second graph', ...])`.

---

<sup>1</sup> [Text in Matplotlib Plots — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/tutorials/text/text_intro.html)

[matplotlib.org/stable/tutorials/text/text\\_intro.html](https://matplotlib.org/stable/tutorials/text/text_intro.html)

<sup>2</sup> [Text properties and layout — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/tutorials/text/text_props.html#sphx-glr-tutorials-text-text-props-py)

[matplotlib.org/stable/tutorials/text/text\\_props.html#sphx-glr-tutorials-text-text-props-py](https://matplotlib.org/stable/tutorials/text/text_props.html#sphx-glr-tutorials-text-text-props-py)

<sup>3</sup> [Annotations — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/tutorials/text/annotations.html#sphx-glr-tutorials-text-annotations-py) [matplotlib.org/stable/tutorials/text/annotations.html#sphx-glr-tutorials-text-annotations-py](https://matplotlib.org/stable/tutorials/text/annotations.html#sphx-glr-tutorials-text-annotations-py)

<sup>4</sup> [matplotlib.pyplot.axis — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axis.html)

[matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.axis.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axis.html)

Remember that these names must be in the same order as you've defined your graphs.

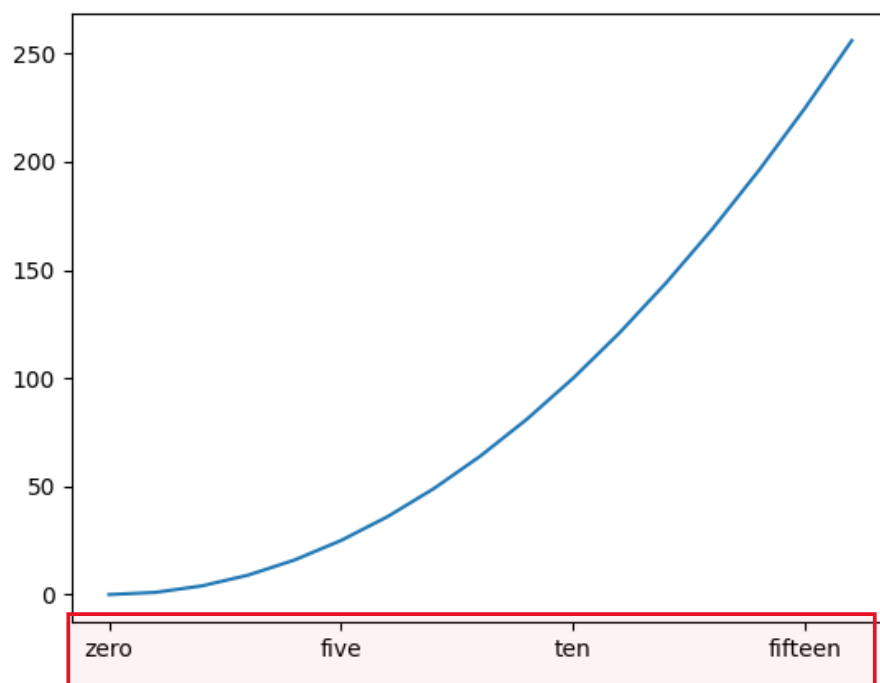
`plt.loglog()` makes a plot with *logarithmic* scaling on both x and y axis.

When we use `plt.xticks(ticks=..... , labels= .....)`, what we choose for *ticks*, will determine the numbers which the plot's x axis is based on; but what we use as *labels*, determines what should be written on x axis. The same method for y axis is `yticks(...)`. This is a way you can show Greek alphabet (like pi) instead of their numeric value on each axis. E.g.:

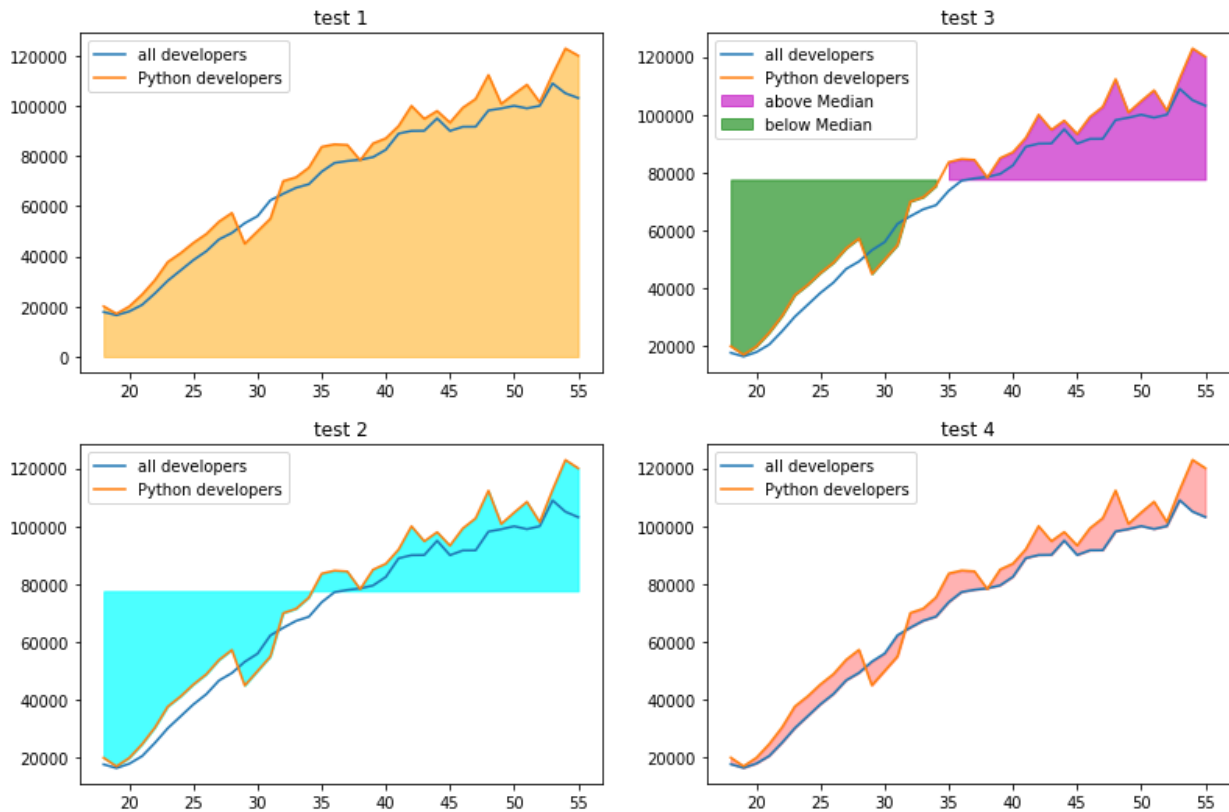
```
import matplotlib.pyplot as plt
import numpy as np

t = np.array(range(0, 17))
s = t ** 2

plt.plot(t, s)
plt.xticks(ticks=[0, 5, 10, 15], labels=['zero', 'five',
'ten', 'fifteen'])
plt.show()
```



We use `plt.fill_between(x-axis, graph, 2nd graph(default = x-axis))` to fill the area between simple line graphs. Give smaller values to *alpha* kwarg to fade the filling color. The examples below represent the results of using this function:



## Color Demo:

You can specify the color and style of your graph lines in *plot()* method. Use *linestyle* kwarg or just a simple str which determines color or(and) line style. To see how, follow the example below: (You'll find a full list of these styles in [Linestyles — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html#linestyles)<sup>1</sup>.)

```
import matplotlib.pyplot as plt
import numpy as np

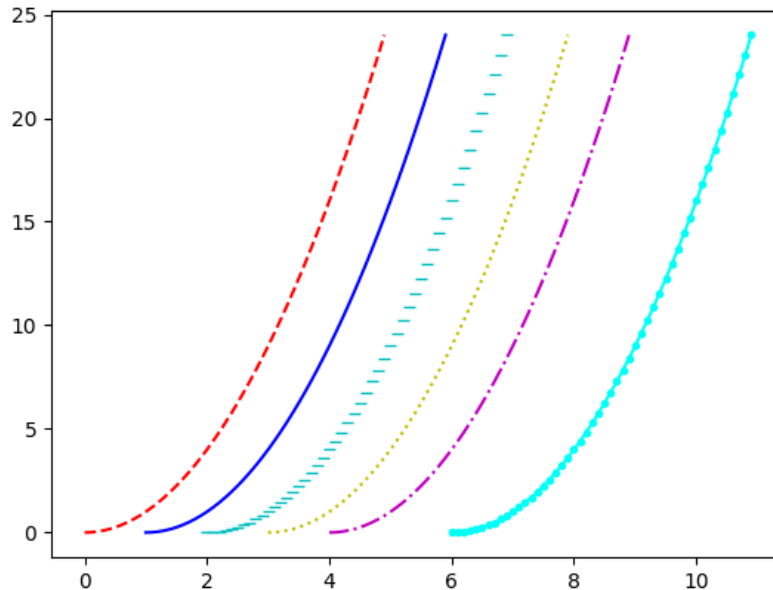
x = np.arange(0, 5, 0.1)
y = x ** 2

plt.plot(x, y, 'r--')
plt.plot(x + 1, y, 'b-')
plt.plot(x + 2, y, 'c_')
plt.plot(x + 3, y, 'y:')
plt.plot(x + 4, y, 'm-.')
# Color Codes for specific colors:
plt.plot(x + 6, y, '#00fff7', marker='.')

plt.show()
```

---

<sup>1</sup> [https://matplotlib.org/stable/gallery/lines\\_bars\\_and\\_markers/linestyles.html#linestyles](https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html#linestyles)



As you see, you can use *color codes* to pick your specific color. In fact, Matplotlib recognizes the following formats to specify a color:

1. an RGB or RGBA tuple of float values in  $[0, 1]$  (e.g. `(0.1, 0.2, 0.5)` or `(0.1, 0.2, 0.5, 0.3)`). RGBA is short for Red, Green, Blue, Alpha;
2. a hex RGB or RGBA string (e.g., `'#0F0F0F'` or `'#0F0F0F0F'`);
3. a shorthand hex RGB or RGBA string, equivalent to the hex RGB or RGBA string obtained by duplicating each character, (e.g., `'#abc'`, equivalent to `'#aabbcc'`, or `'#abcd'`, equivalent to `'#aabbccdd'`);
4. a string representation of a float value in  $[0, 1]$  inclusive for gray level (e.g., `'0.5'`);
5. a single letter string, i.e. one of `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`, which are short-hand notations for shades of blue, green, red, cyan, magenta, yellow, black, and white;
6. a X11/CSS4 ("html") color name, e.g. `"blue"`;
7. a name from the [xkcd color survey](https://xkcd.com/color/rgb/)<sup>1</sup>, prefixed with `'xkcd:'` (e.g., `'xkcd:sky blue'`);

---

<sup>1</sup> <https://xkcd.com/color/rgb/>



8. a "Cn" color spec, i.e. 'C' followed by a number, which is an index into the default property cycle ([rcParams\["axes.prop\\_cycle"\]](#)<sup>1</sup>(default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`)); the indexing is intended to occur at rendering time, and defaults to black if the cycle does not include color.
9. one of {'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'} which are the Tableau Colors from the 'tab10' categorical palette (which is the default color cycle).<sup>2</sup>

review all these methods with this instance:<sup>3</sup>

```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0.0, 2.0, 201)
s = np.sin(2 * np.pi * t)

# 1) RGB tuple:
fig, ax = plt.subplots(facecolor=(.18, .31, .31))
# 2) hex string:
ax.set_facecolor('#eafff5')
# 3) gray level string:
ax.set_title('Voltage vs. time chart', color='0.7')
# 4) single letter color string
ax.set_xlabel('time (s)', color='c')
# 5) a named color:
ax.set_ylabel('voltage (mV)', color='peachpuff')
# 6) a named xkcd color:
ax.plot(t, s, 'xkcd:crimson')
# 7) Cn notation:
ax.plot(t, .7 * s, color='C4', linestyle='--')
```

---

<sup>1</sup> [Customizing Matplotlib with style sheets and rcParams — Matplotlib 3.5.2 documentation](#)

[https://matplotlib.org/stable/tutorials/introductory/customizing.html?highlight=axes.prop\\_cycle#a-sample-matplotlibrc-file](https://matplotlib.org/stable/tutorials/introductory/customizing.html?highlight=axes.prop_cycle#a-sample-matplotlibrc-file)

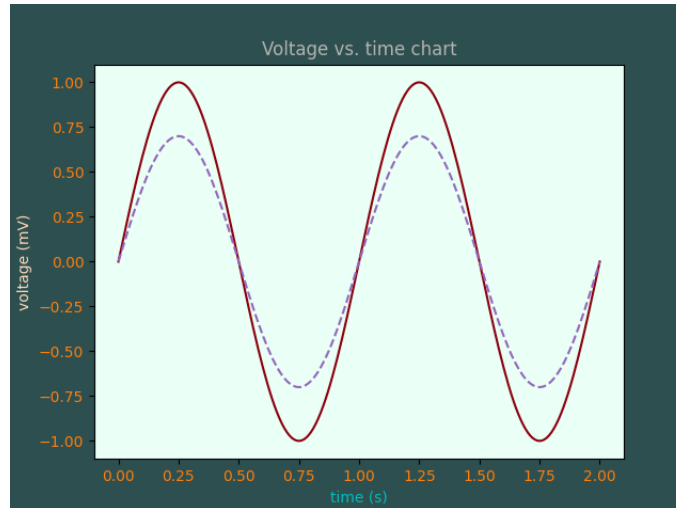
<sup>2</sup> [Matplotlib documentation — Matplotlib 3.5.2 documentation](#)

<https://matplotlib.org/stable/index.html>

<sup>3</sup> [Matplotlib documentation — Matplotlib 3.5.2 documentation](#)

<https://matplotlib.org/stable/index.html>

```
# 8) tab notation:  
ax.tick_params(labelcolor='tab:orange')  
  
plt.show()
```



you can use the site [HTML Color Codes](https://htmlcolorcodes.com/)<sup>1</sup>, to find your desired color's code.

In addition to line customization, you can use **markers** to mark specific points. See a full list of markers and their usage examples at: [matplotlib.markers — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/api/markers_api.html)<sup>2</sup>.

---

<sup>1</sup> <https://htmlcolorcodes.com/>

<sup>2</sup> [https://matplotlib.org/stable/api/markers\\_api.html](https://matplotlib.org/stable/api/markers_api.html)

## r-strings:

We've learnt about *f-strings* in Input-Output headline. That was used for formatting the strings. Here, however, we intend to use special chars and fonts in strings. This might be done by *r-strings* that use **LaTeX** font for scientific and mathematical expressions and chars.

LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents.<sup>1</sup>



There are tons of special chars and signs you can add for scientific data, using r-string; have a look at [Writing mathematical expressions — Matplotlib 3.5.2 documentation](#)<sup>2</sup> for a thorough list.

To use these chars, follow these instructions to make a r-string:

- Place a *r* before string (as we placed *f* for f-strings).
- Place a dollar sign (\$) inside the string at both beginning and the end of it.
- Note the usage of braces and backslashes.

There are so many examples at the reference site mentioned above.

Here is one: `r'$\frac{5 - \frac{1}{x}}{4}$'` → 
$$\frac{5 - \frac{1}{x}}{4}$$

---

<sup>1</sup> [LaTeX - A document preparation system \(latex-project.org\)](https://www.latex-project.org/)

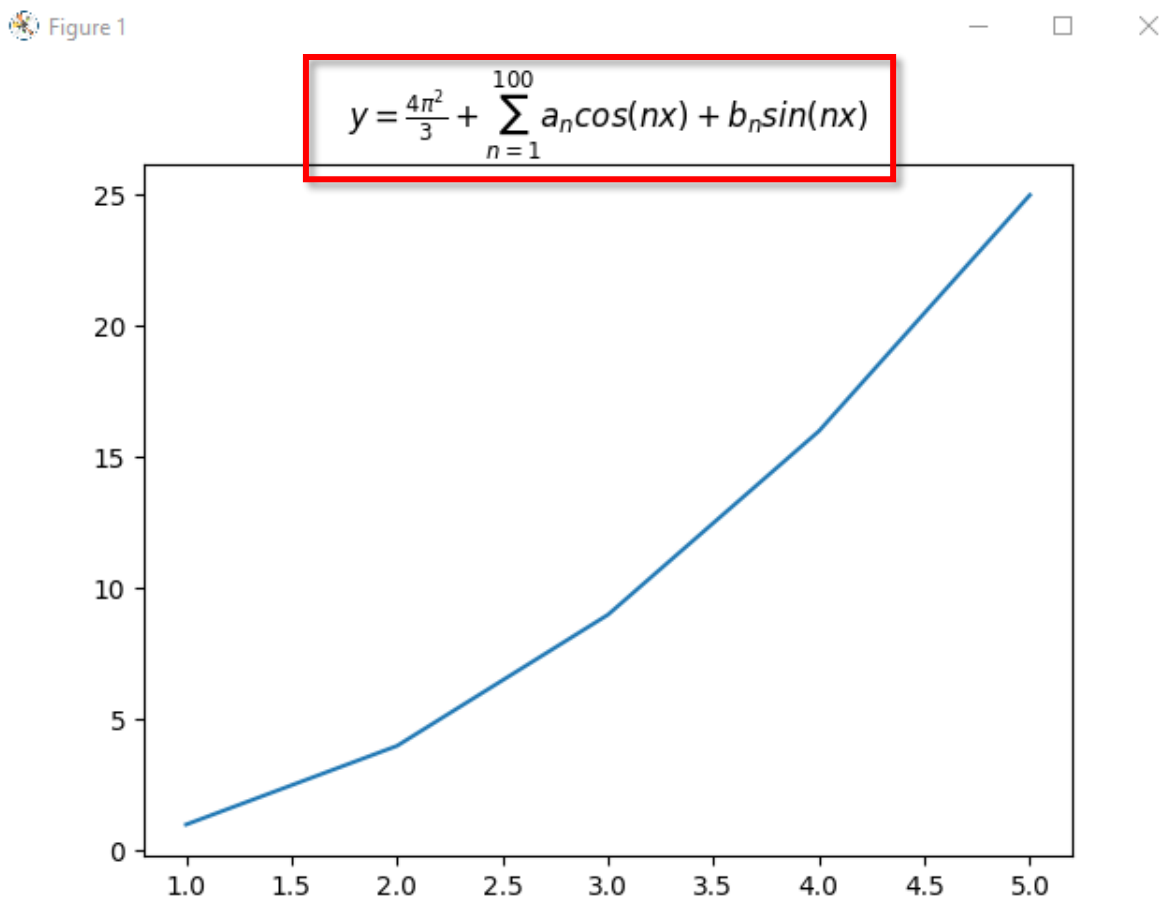
<https://www.latex-project.org/>

<sup>2</sup> <https://matplotlib.org/stable/tutorials/text/mathtext.html>

You can also merge f-string and r-string as *fr-string*, but it's very important to note the number of braces '{}' you've got to use to distinguish the usage of phrases belonging to f-string from those of r-string. This might be a little complicated at first sight. Have an example for practice:

```
x = range(1, 6)
y = [n * n for n in x]
end = 100

plt.plot(x, y)
plt.title(fr"$y=\frac{4\pi^{{{2}}}}{{{3}}} + \sum_{{{n=1}}}^{{{end}}} a_{{{n}}} \cos(nx) + b_{{{n}}} \sin(nx)$")
plt.show()
```



## Multiple Plots:

As it's been referred to, if you want to show multiple graphs on one plot, define them all in one line using `plot(x, graph1, graph2, ...)` or in different lines, like `plot(x, graph_no.)` as we did before. Finally use the `plt.show()` method only once in the end to reveal the plot.

However, if you want to have multiple plots on your figure, you must use **subplots** like: `fig, (ax1, ax2, ...) = plt.subplots(nrows=..., ncols= ...)`

You can also declare figure's size by '`figsiz=(width, height)`' kwarg.

You're also able to create more than one figure. Consequently, you'll see more than one window when you run the code. E.g.:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.1)
y1 = x ** 2
y2 = np.sqrt(x)
y3 = np.exp(x)
y4 = np.log(x)
y5 = np.sin(x)
y6 = np.arcsin(x)

fig1, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
sharex=True)
fig2, (ax3, ax4) = plt.subplots(nrows=1, ncols=2,
sharey=False)
fig3, ax5 = plt.subplots(nrows=1, ncols=1)

ax1.plot(x, y1)
ax2.plot(x, y2)
ax3.plot(x, y3)
ax4.plot(x, y4)
ax5.plot(x, y5, label=r'$y=\mathrm{sin}(x)$')
ax5.plot(x, y6, label=r'$y=\mathrm{arcsin}(x)$')

ax5.legend()
```

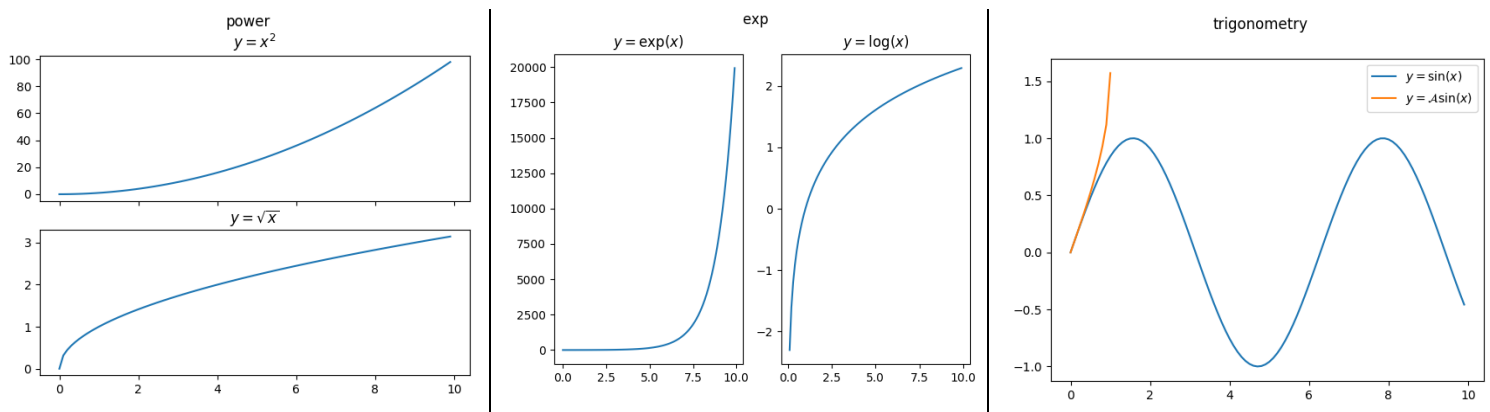
```

ax1.set_title(r'$y=x^{2}$')
ax2.set_title(r'$y=\sqrt{x}$')
ax3.set_title(r'$y=\exp(x)$')
ax4.set_title(r'$y=\log(x)$')

fig1.suptitle('power')
fig2.suptitle(r'$\exp$')
fig3.suptitle('trigonometry')

plt.show()

```



Another kwarg for *subplots* method is *gridspec\_kw*. You can use it to customize subplots' size and location. This **\*\*kwarg** must be assigned with a dict like: *gridspec\_kw* = {'width\_ratios': [..., ...], 'height\_ratios': [..., ...], 'wspace': ..., 'hspace': ...}

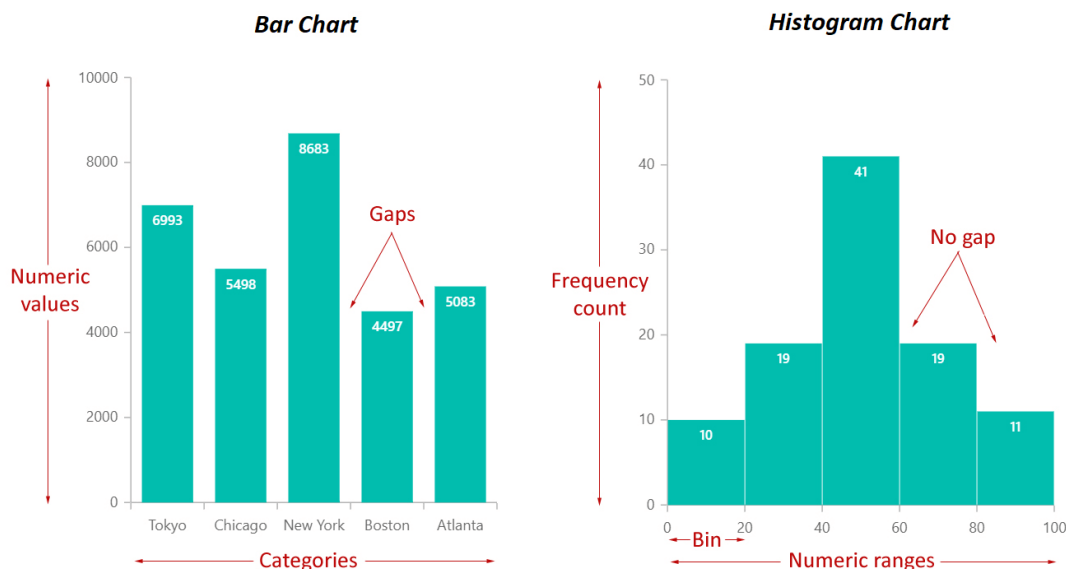
These dict keys are meant to:

- **width\_ratios**: set width size of each subplot.
- **height\_ratios**: set height size of each subplot.
- **wspace**: give “wspace” amount of space vertically to separate the subplots.
- **hspace**: give “hspace” amount of space horizontally to separate the subplots.

You'll find more examples at: [How to Create Different Subplot Sizes in Matplotlib? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-create-different-subplot-sizes-in-matplotlib/)<sup>1</sup>

## BAR GRAPHS AND HISTOGRAMS

In addition to simple line graphs that we used to draw with `plot()`, there are other types of graphs Matplotlib provides. Bar graphs and Histograms are two of the most common ones. These two, despite their similarities, are actually different. See below<sup>2</sup>:



<sup>1</sup> <https://www.geeksforgeeks.org/how-to-create-different-subplot-sizes-in-matplotlib/>

<sup>2</sup> [8 key differences between Bar graph and Histogram chart | Syncfusion](https://www.syncfusion.com/blogs/post/difference-between-bar-graph-and-histogram-chart.aspx)

<https://www.syncfusion.com/blogs/post/difference-between-bar-graph-and-histogram-chart.aspx>

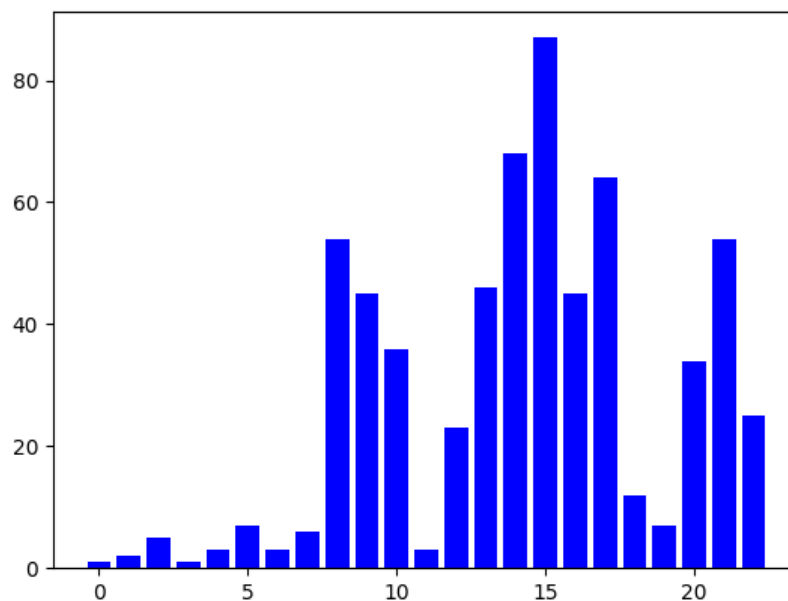
To make a bar chart, just use `plt.bar(.....)` instead of `plt.plot()`. For horizontal bar chart, use `plt.barh(.....)`.

In addition to *label*, *color* is another kwarg you can use to customize your diagram; so is *rwidth*.

Simplest way to produce a histogram/bar chart is:

```
population = [1, 2, 5, 1, 3, 7, 3, 6, 54,
              45, 36, 3, 23, 46, 68, 87,
              45, 64, 12, 7, 34, 54, 25]
ids = [x for x in range(len(population))]

plt.bar(ids, population, color='b')
plt.show()
```



However, a better and more professional way is to create *bins*, which are the sections to which we divide our data (in the horizontal axis). Use `plt.hist(.....)` instead of `bar` now. E.g.:



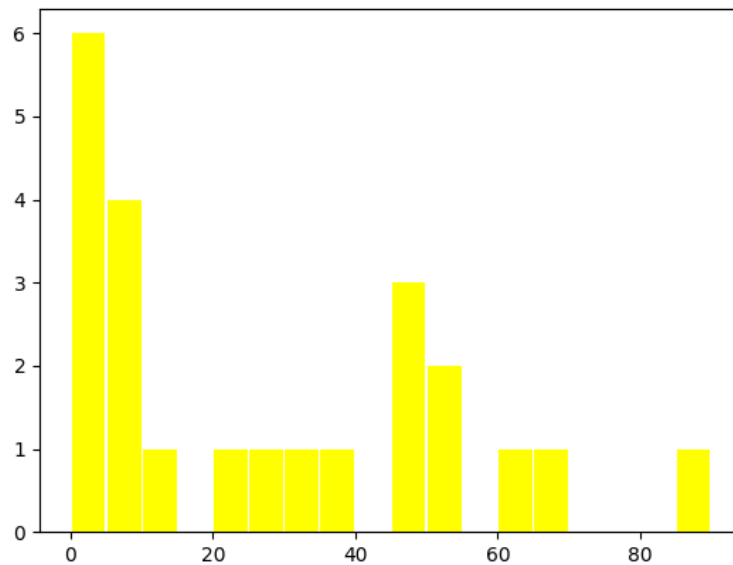
```

population = [1, 2, 5, 1, 3, 7, 3, 6, 54,
              45, 36, 3, 23, 46, 68, 87,
              45, 64, 12, 7, 34, 54, 25]

bins = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45,
        50, 55, 60, 65, 70, 75, 80, 85, 90]

plt.hist(population, bins, histtype='bar', \
         color='yellow', rwidth=0.95)
plt.show()

```



## SCATTER PLOTS

Scatter plot is a diagram based on dots. Use `plt.scatter(.....)` to produce one. Two useful `**kwargs` used here are `s` (size) and `marker`. E.g.:

```

import matplotlib.pyplot as plt
import numpy as np

```

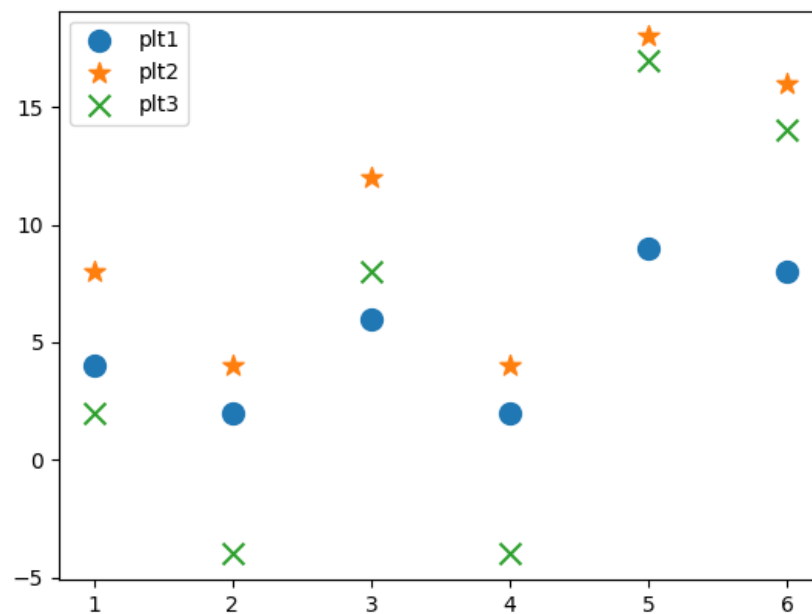
```

x = [range(1, 7)]
y = [4, 2, 6, 2, 9, 8]
y2 = np.array(y) * 2
y3 = np.array(y) * 3 - 10

plt.scatter(x, y, s=100, marker='o', label='plt1')
plt.scatter(x, y2, s=100, marker='*', label='plt2')
plt.scatter(x, y3, s=100, marker='x', label='plt3')

plt.legend()
plt.show()

```



You can add *cmaps* (colour-maps) as a legend for these kinds of plots. Find more info at: [Choosing Colormaps in Matplotlib — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/tutorials/colors/colormaps.html)<sup>1</sup>

<sup>1</sup> <https://matplotlib.org/stable/tutorials/colors/colormaps.html>

## STACK PLOTS

Stack charts are used to clarify the share of different subjects in a plot. For this kind, use `plt.stackplot(.....)`

To add legend \_which is necessary for these charts\_, we have to perform a trick. By defining empty plots, with different \_but specific\_ colors and names, we create a proper legend for our plot. Learn this trick better in the example below, where we draw the share of daily activities done in 5-weekdays:

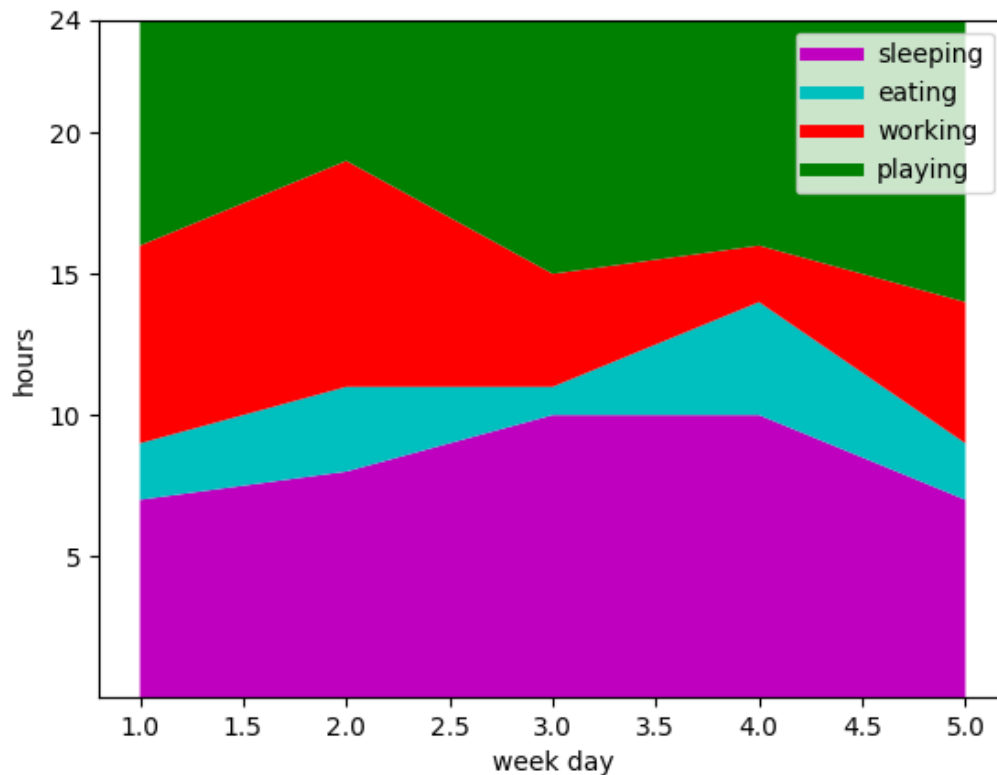
```
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5]
sleeping = [7, 8, 10, 10, 7]
eating = [2, 3, 1, 4, 2]
working = [7, 8, 4, 2, 5]
playing = [8, 5, 9, 8, 10]

plt.plot([], [], color='m', label='sleeping', \
         linewidth=5)
plt.plot([], [], color='c', label='eating', \
         linewidth=5)
plt.plot([], [], color='r', label='working', \
         linewidth=5)
plt.plot([], [], color='g', label='playing', \
         linewidth=5)

plt.stackplot(days, sleeping, eating, working, \
              playing, colors=['m', 'c', 'r', 'g'])

plt.xlabel('week day')
plt.ylabel('hours')
plt.ylim(0, 24)
plt.yticks(ticks=[5, 10, 15, 20, 24])
plt.legend()
plt.show()
```



## PIE CHARTS

Pie charts, as one of the most common and practical plot types worldwide, are used to show the percentage different objects share of a total value.

Use `plt.pie(iterable, labels=..., colors=..., explode=..., autopct=..., shadow=...)`

The *numerical list*, *labels*, *colors*, and *explosion* values must be determined with the mentioned sequence. Increase the *explosion* value

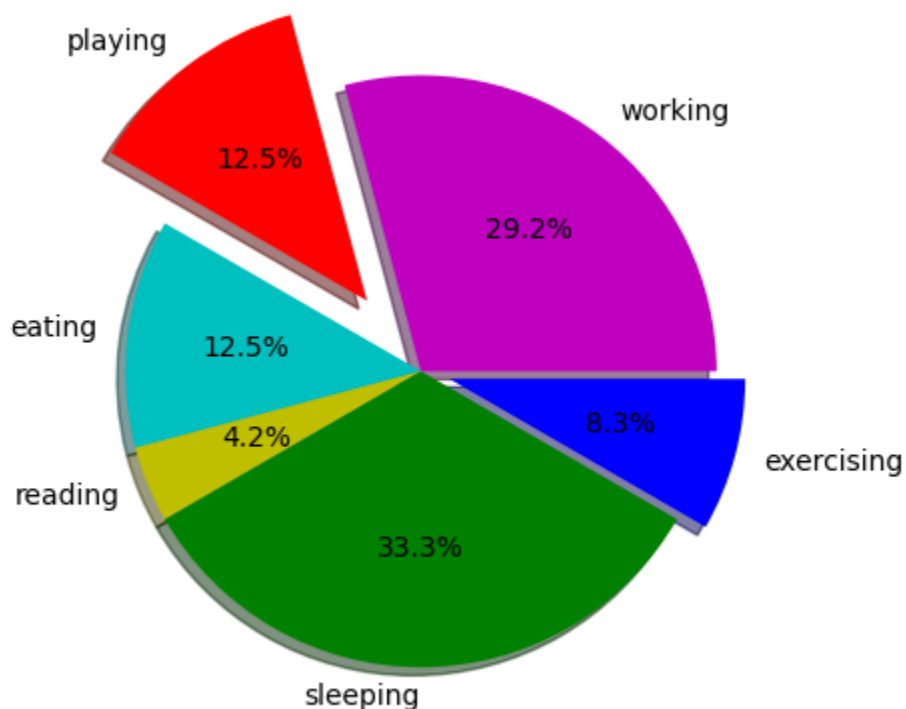
of each section to have it more separated from the main circle. Use *autopct* for automatic percentage calculation. Use *shadow* to make the diagram more likely to 3D mode. You can remark a thorough list of possible **\*\*kwargs** for this plot in the appendix.

```
import matplotlib.pyplot as plt

hours = [7, 3, 3, 1, 8, 2]
acts = ['working', 'playing', 'eating',
        'reading', 'sleeping', 'exercising']

plt.pie(hours, labels=acts,
        colors=['m', 'r', 'c', 'y', 'g', 'b'],
        explode=[0, 0.3, 0, 0, 0, 0.1],
        autopct='%1.1f%%', shadow=True)

plt.show()
```



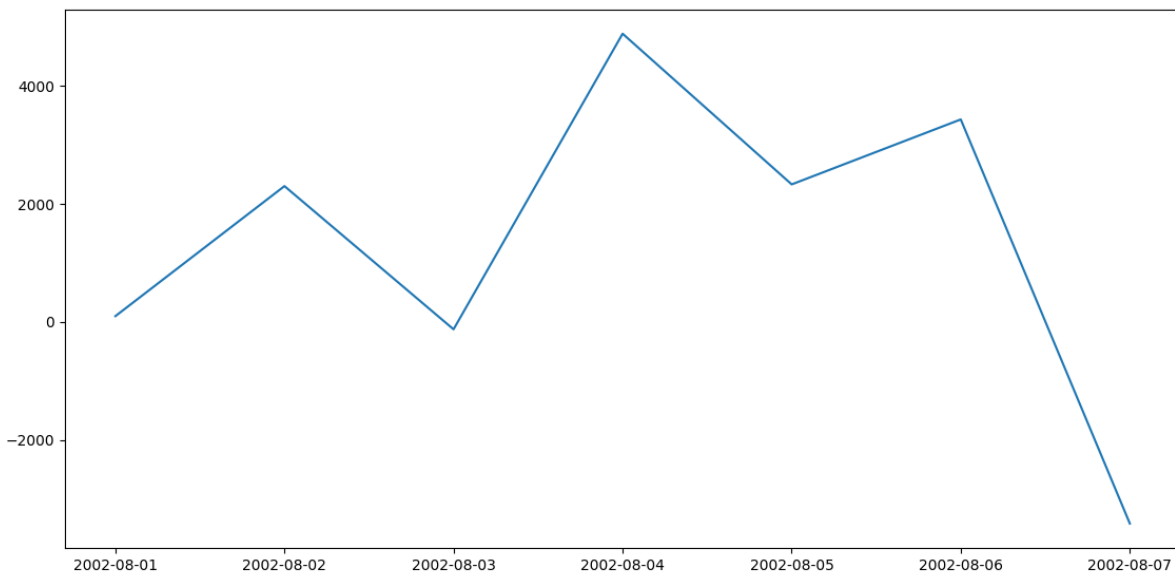
# TIME SERIES

This kind of plots is usually used to show the quality of something's alternation based on Time. One way to create one, is:

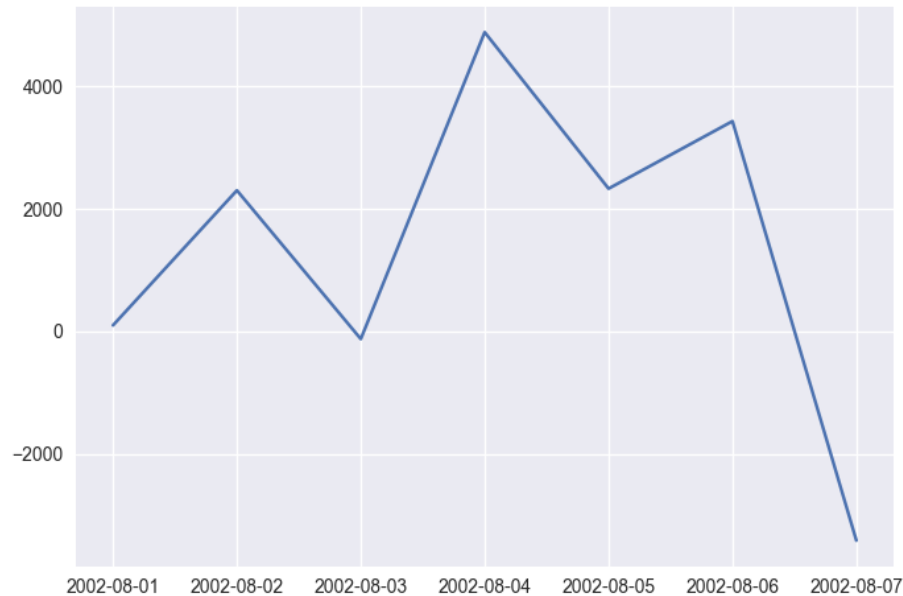
```
import matplotlib.pyplot as plt
import datetime as dt

# plt.style.use('seaborn')
days = [
    dt.date(2002, 8, 1),
    dt.date(2002, 8, 2),
    dt.date(2002, 8, 3),
    dt.date(2002, 8, 4),
    dt.date(2002, 8, 5),
    dt.date(2002, 8, 6),
    dt.date(2002, 8, 7)
]

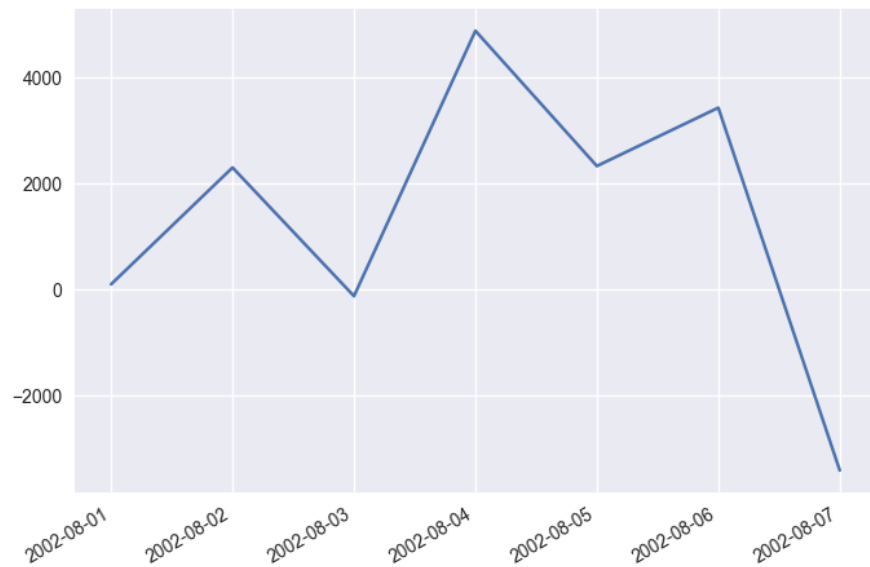
profit = [100, 2302, -123, 4883, 2331, 3431, -3412]
plt.plot(days, profit)
plt.show()
```



And if we UNCOMMENT the 3<sup>rd</sup> line, the result will be:



Use the order `plt.gcf().autofmt_xdate()`<sup>1</sup> to beautify x axis format like this:



---

<sup>1</sup> *gcf* stands for *get current figure*.

In order to use months' names instead of numbers, we refactor our code to this:

```
import matplotlib.pyplot as plt
from matplotlib import dates as mpl_dates
import datetime as dt

plt.style.use('seaborn')
days = [
    dt.date(2002, 8, 1),
    dt.date(2002, 8, 2),
    dt.date(2002, 8, 3),
    dt.date(2002, 8, 4),
    dt.date(2002, 8, 5),
    dt.date(2002, 8, 6),
    dt.date(2002, 8, 7)
]

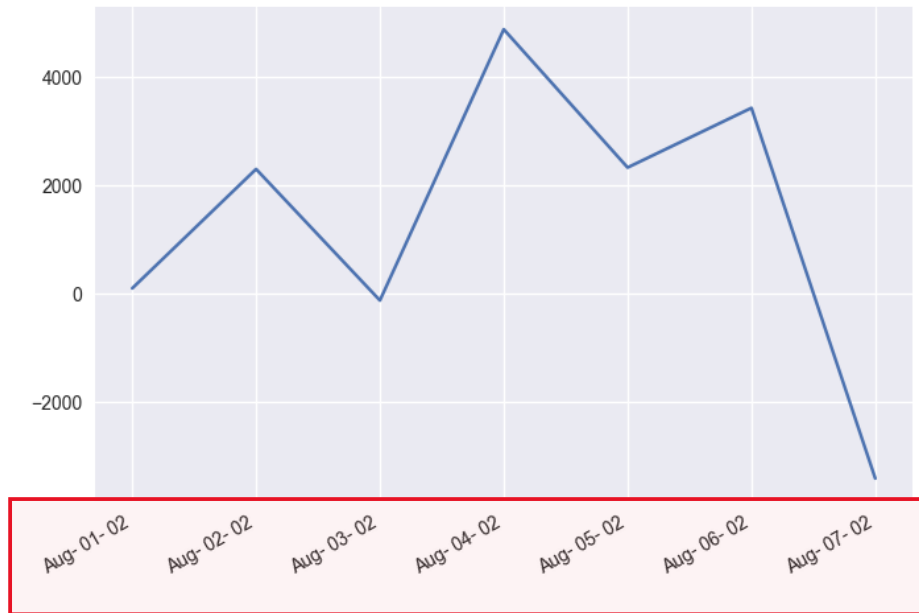
profit = [100, 2302, -123, 4883, 2331, 3431, -3412]

# altering format to our desire as below:
date_format = mpl_dates.DateFormatter('%b- %d- %y')
# b stands for month, d for day and y for year

plt.gca().xaxis.set_major_formatter(date_format)

plt.plot(days, profit)
plt.gcf().autofmt_xdate()
plt.show()
```





There are dozens of other customizations for these graphs; for sorting, visualizing and other things you can search for.

## LIVE PLOTS

Live plots are the ones that change every moment. The result of these plots is a dynamic animation, rather than static images. Here is an example of creating live plots:

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation as fa
import itertools as it
import numpy as np

plt.style.use('Solarize_Light2')
```

```

x = []
y = []

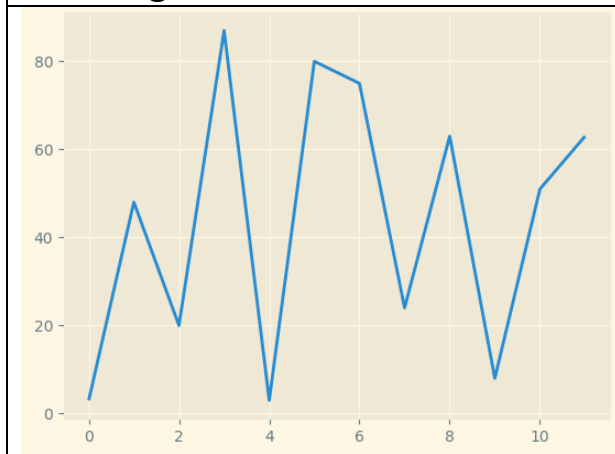
index = it.count()

def animate(i):
    x.append(next(index))
    y.append(np.random.randint(1, 100))
    plt.cla()
    plt.plot(x, y)

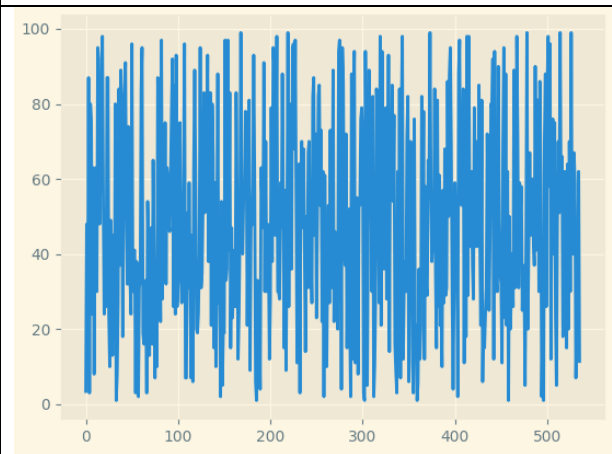
ani = fa(plt.gcf(), animate, interval=100)
# interval determines refresh rate of plot,
# based on milli second
plt.show()

```

At first glances:



After a few seconds:



## OTHER PLOTS

As an open source, widely praised module, Matplotlib provides dozens of other kinds of fascinating diagrams. Take a glance at the official website for further info: [Matplotlib — Visualization with Python](https://matplotlib.org/)<sup>1</sup>

Here are a few examples of some other types of plots<sup>2</sup>:

---

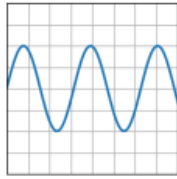
<sup>1</sup> <https://matplotlib.org/>

<sup>2</sup> [Plot types — Matplotlib 3.5.2 documentation](https://matplotlib.org/stable/plot_types/index.html)

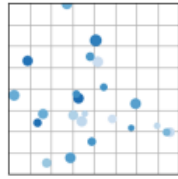
[https://matplotlib.org/stable/plot\\_types/index.html](https://matplotlib.org/stable/plot_types/index.html)

## Basic

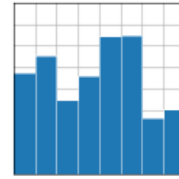
Basic plot types, usually y versus x.



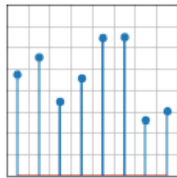
`plot(x, y)`



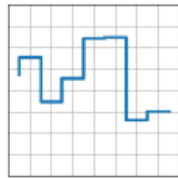
`scatter(x, y)`



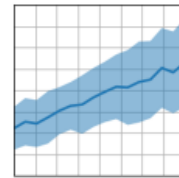
`bar(x, height) / barh(y, width)`



`stem(x, y)`



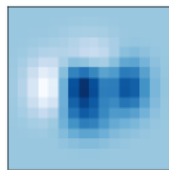
`step(x, y)`



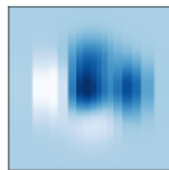
`fill_between(x, y1, y2)`

## Plots of arrays and fields

Plotting for arrays of data  $z(x, y)$  and fields  $u(x, y)$ ,  $v(x, y)$ .



`imshow(Z)`



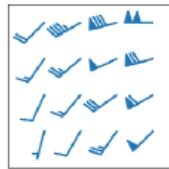
`pcolormesh(X, Y, Z)`



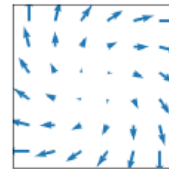
`contour(X, Y, Z)`



`contourf(X, Y, Z)`



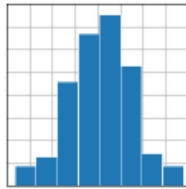
`barbs(X, Y, U, V)`



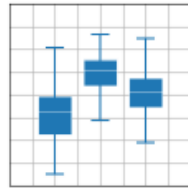
`quiver(X, Y, U, V)`

# Statistics plots

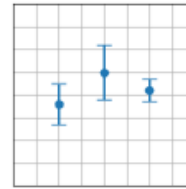
Plots for statistical analysis.



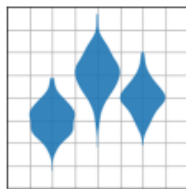
hist(x)



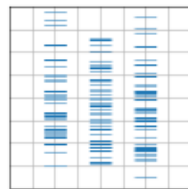
boxplot(X)



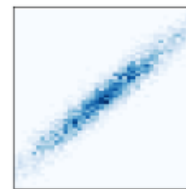
errorbar(x, y, yerr, xerr)



violinplot(D)



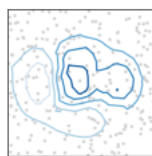
eventplot(D)



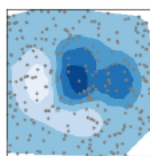
hist2d(x, y)

## Unstructured coordinates

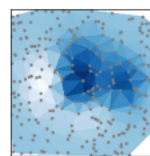
Sometimes we collect data  $z$  at coordinates  $(x, y)$  and want to visualize as a contour. Instead of gridding the data and then using `contour`, we can use a triangulation algorithm and fill the triangles.



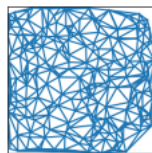
tricontour(x, y, z)



tricontourf(x, y, z)



tripcolor(x, y, z)

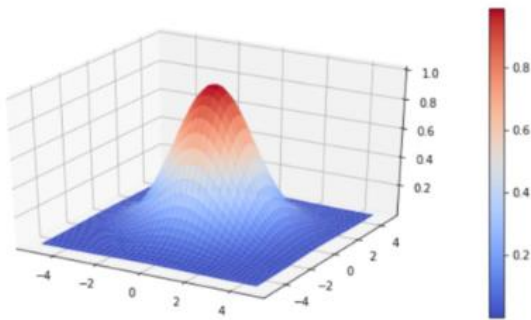


triplot(x, y)

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 X = np.arange(-5, 5, 0.25)
5 Y = np.arange(-5, 5, 0.25)
6 X, Y = np.meshgrid(X, Y)
7 Z = np.exp(-(X**2+Y**2)/5)
8
9 fig = plt.figure(figsize=(10,5))
10 ax = fig.gca(projection='3d')
11 surf = ax.plot_surface(X, Y, Z, cmap="coolwarm")
12 plt.colorbar(surf)
13 plt.show()
14

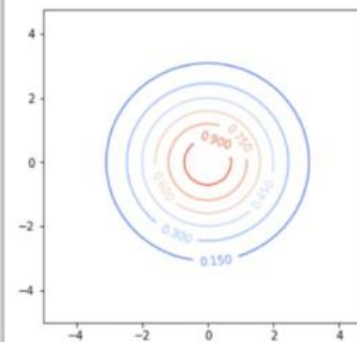
```



```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(-5, 5, 0.25)
5 y = np.arange(-5, 5, 0.25)
6 X, Y = np.meshgrid(x, y)
7 Z = np.exp(-(X**2+Y**2)/5)
8
9 fig = plt.figure(figsize=(5,5))
10 cs = plt.contour(X, Y, Z, cmap="coolwarm")
11 plt.clabel(cs)
12 plt.show()
13

```



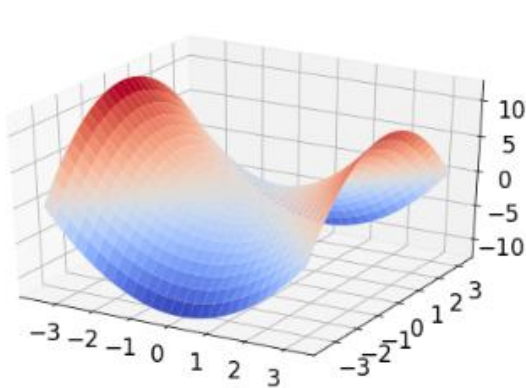
```

import matplotlib.pyplot as plt
import numpy as np

X = np.arange(-3.5, 3.5, 0.25)
Y = np.arange(-3.5, 3.5, 0.25)
X, Y = np.meshgrid(X, Y)
Z = (X**2 - Y**2)

fig = plt.figure(figsize=(10,5))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap="coolwarm")
plt.colorbar(surf)
plt.show()

```



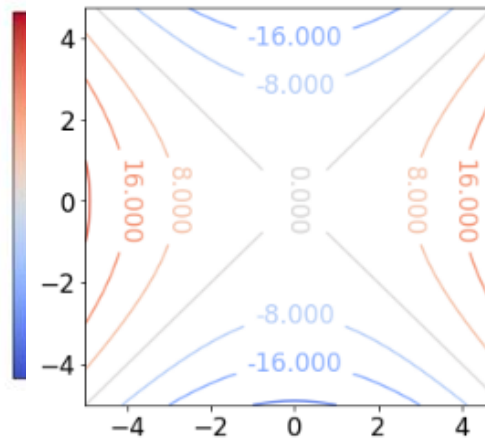
```

import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-5, 5, 0.25)
y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2

fig = plt.figure(figsize=(5,5))
cs = plt.contour(X, Y, Z, cmap="coolwa")
plt.clabel(cs)
plt.show()

```





# Csv

This module is dedicated to reading and writing to files. See an example to learn how it must be used:

```
import numpy as np
import matplotlib.pyplot as plt
import csv

x, y = [], []
with open('test.txt', 'r') as myfile:
    data = csv.reader(myfile, delimiter='-')
    # delimiter is the char by whcih we've
    # seperated our coloumns in our .txt file

    for what in data:
        x.append(int(what[0]))
        y.append(int(what[1]))

print(x)
print(y)
plt.plot(x, y)
plt.show()
```

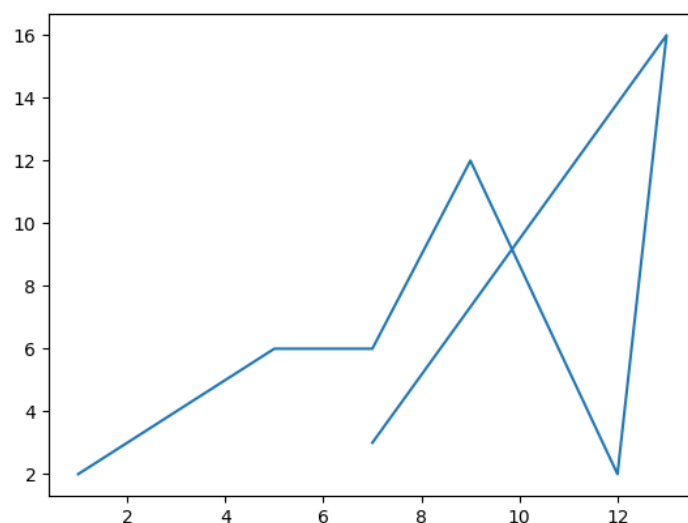
## Console:

```
[1, 3, 5, 7, 9, 12, 13, 7]
[2, 4, 6, 6, 12, 2, 16, 3]
```

## test.txt:

```
1-2
3-4
5-6
7-6
9-12
12-2
13-16
7-3
```

## Figure:





And here comes an example about how to write in files:

```
import csv

header = ['col.1', 'col.2', 'col.3']
x = 11
y = 22
z = 33

l1, l2, l3 = [], [], []

# w+ clears (truncates) file's previous content,
# and replaces its data with a new one:
with open('Bit.csv', 'w+') as cFile:
    csv_writer = csv.DictWriter(cFile, header)
    csv_writer.writeheader()

for i in range(10):
    with open('Bit.csv', 'a') as cFile:
        csv_writer = csv.DictWriter(cFile, header)

        info = {
            'col.1': x,
            'col.2': y,
            'col.3': z
        }
        csv_writer.writerow(info)
        x = x * 2
        y = y + x / 3
        z = z + 20 - y

with open('Bit.csv', 'r') as cFile:
    data = csv.reader(cFile)
    d = list(data)
    myHeaders = d[0]
    for what in d:
        try:
            l1.append(float(what[0]))
            l2.append(float(what[1]))
            l3.append(float(what[2]))
        except:
```

```

pass

print(f'{myHeaders[0]}\t{myHeaders[1]}\t{myHeaders[2]}')
for i in range(len(l1)):
    print(f'{round(float(l1[i]), 2)}\t'
          f'\t{round(float(l2[i]), 2)}\t'
          f'\t{round(float(l3[i]), 2)}')

```

### Console:

```

col.1  col.2  col.3
11.0   22.0   33.0
22.0   29.33  23.67
44.0   44.0   -0.33
88.0   73.33  -53.67
176.0  132.0  -165.67
352.0  249.33 -395.0
704.0  484.0 -859.0
1408.0 953.33 -1792.33
2816.0 1892.0 -3664.33
5632.0 3769.33 -7413.67

```

### Bit.csv before running:

	A
1	time←price
2	2020-5-18←1250
3	2020-5-24←8710
4	2020-5-26←2461
5	2020-5-23←1732
6	2020-5-22←5127
7	2020-5-19←1356
8	2020-5-20←851
9	2020-5-25←5472
10	2020-5-21←1436

### Bit.csv after running:

col.1	col.2	col.3
11	22	33
22	29.33333	23.66667
44	44	-0.33333
88	73.33333	-53.6667
176	132	-165.667
352	249.3333	-395
704	484	-859
1408	953.3333	-1792.33
2816	1892	-3664.33
5632	3769.333	-7413.67

# Pandas

In addition to CSV and NUMPY methods for working with files, pandas module can do the deed as well. Pandas can even distinguish the header. E.g.:

```
import pandas as pd

data = pd.read_csv('test.txt')
print(data)
ages = data['age']
saleies = data['sal']
number_of_children = data['NoC']
```

Console:	test.txt:
age sal NoC	age,sal,NoC
0 80 5 3	80,5,3
1 56 7 2	56,7,2
2 54 9 3	54,9,3
3 76 19 6	76,19,6
4 90 2 9	90,2,9
5 36 9 1	36,9,1
6 23 7 0	23,7,0

Keep in mind that there shouldn't be any *space* after comma in header row at the *test.txt* file.

Use [pd.read\\_excel\(\)](#) for *xls* or *xlsx* files.



# Minor Modules

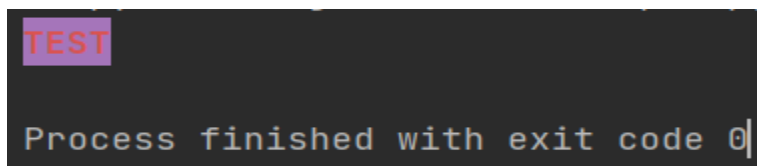
## TIME

You can access everything related to time with this module. [time\(\)](#) function, for example, returns current moment.

## TERMCOLOR

This module is used to customize what you print. E.g.:

```
from termcolor import *  
  
cprint('TEST', 'magenta',  
       'on_red', ['bold', 'reverse'])
```



```
TEST  
  
Process finished with exit code 0
```

## PYFIGLET

This package is used for printing in a beautiful shape called ASCII-ART, module **printy** does the same thing, but more complicated. Further info at [printy · PyPI](#)<sup>1</sup>. Best syntax for pretty-printing in *pyfiglet* is:

[\*figlet\\_format\(text, font='.....'\)\*](#)

---

<sup>1</sup> <https://pypi.org/project/printy/>

(For its font `**kw`, you must use a pre-figured font of this module, which you can find a full list of very amusing ones in [FIGlet - hosted by PLiG](http://www.figlet.org/examples.html)<sup>1</sup>

## DATETIME

There are some practical methods in this module you might need:

- `date(yr, mn, dy)`
- `time(hr, min, sc,  $\mu$ sc)`
- `datetime(yr, mn, dy, hr, min, sc,  $\mu$ sc)`

See a small usage:

```
import datetime as dt

myBirthday = dt.datetime(2002, 8, 1, 12, 15, 30, 50)
print(myBirthday)
```

2002-08-01 12:15:30.000050

## MATH

A useful module for mathematicians and anything related to math, from special constants \_like  $\Pi$ ,  $e$ , ...\_ to mathematical functions and all.

---

<sup>1</sup> <http://www.figlet.org/examples.html>

# Virtual Environment

After learning about different modules, you might want to install some packages on your Python. Gradually, the number of installed packages might go insane, and with all these burdens on your Python Interpreter core, it will appreciably slow down. To prevent this, we use virtual environments (venv).

In fact, for every project of yours, you'll create a venv of its own, in its directory. This will make a copy of raw Python in your venv. Thereafter, when you work on your project and want to install some packages, they will be installed on the duplicate version of Python on your specific venv, not on the main Python core. In this case, you can access modules of every venv you created, on their own projects. Therefore, your main interpreter won't be burdened.

To create a venv, first create your directory folder wherever you want. Then go to the terminal (CMD) and type:

***Python -m venv directory\_name***

(This works for Windows and Mac OS, for Linux, you should write *Python3* instead of *Python*)

Use directory's path if your terminal is not currently set on THE path.

```
D:\myEnv>python -m venv myEnv
```

```
C:\Users\Mohammad\Desktop>python -m venv D:\myEnv
```

With this, the venv will copy some folders in your directory. To activate the venv, based on your OS, type these commands:



Windows:

***Directory\_name\_or\_address\Scripts\activate.bat***

```
D:\>myEnv\Scripts\activate.bat  
(myEnv) D:\>_
```



Linux or



Mac:

***source Directory\_name/bin/activate***

Thereafter, you see your venv directory name on your terminal on the left side of every command line.

Then, if you try, you'll see almost no package installed on your new venv because it's totally raw. So you can use it to install whatever you need on your project from now on, without troubling other venvs.

```
(myEnv) D:\>pip list  
Package      Version  
-----  
pip          20.2.3  
setuptools   49.2.1
```

You can find more guides if need be, at [12. Virtual Environments and Packages — Python 3.10.6 documentation](https://docs.python.org/3/tutorial/venv.html#virtual-environments-and-packages)<sup>1</sup>

---

<sup>1</sup> <https://docs.python.org/3/tutorial/venv.html#virtual-environments-and-packages>

# Graphical User Interference

We've been seeing the result of whatever we've been working with in Terminal or Console up to now; but here we intend to learn some elementary basics of graphical interfaces.

There are multiple frameworks for Python GUI such as:

- PyQt5.
- Tkinter.
- Kivy.
- wxPython.
- Libavg.
- PySimpleGUI.
- PyForms.
- Wax.

Here we learn more about **tkinter**, which is a simple pre-installed Python GUI framework.

We have to import all that is in this package like: `from tkinter import *`

After importing, we create a graphical page and save it in a var like:  
`var = Tk()`

Use `var.title("new_title")` to entitle your page.

Use `var.geometry('width x height')` to configure initial size of the window.

Use `var.resizable(width = False, height = False)` to lock the dimensions of the window.

Use `var.mainloop()` to open up the page.

Use `var.configure(bg = 'background_color')` to change the window's background color.



There are some useful widgets you can put in your GUI: **Labels** are strings you can show on your page, **Entries**, input boxes with which you can receive data from client, and **Buttons** used to do special functionalities when clicked.

Use `label_name = Label('page_name', text = 'str_to_be_written', **kw)` to create labels.

To have the label printed on the page, you have to clarify its position like: `label_name.place(x= ....., y= .....,)`

Use `input_name = Entry('page name')` to take input in the window.

Like labels, you have to clarify the position of the blank space like: `input_name.place(x= ....., y= .....,)`

You can have access to what client has entered in input box by using `input_name.get()` later.

In addition, you can save client's data in a var, by giving it to a `**kwarg` called 'textvariable': `input_name = Entry('page name', textvariable = ...)`

To this `textvariable` part, we assign a variable we've defined before as **StringVar** or **IntVar**, so the value the client enters, will be saved in that.

We access our variable later by `variable_name.get()` method.

`StringVar` and `IntVar` are just data types that stand for str and int, only specific to `tkinter`. That's why we save input data in them, not in simple Python var types.

To create a var in these types, first, we introduce them like:

`myVar = StringVar()`

And we assign data to them later, like: `myVar.set(.....)`

We can use `textvariable = StringVar or IntVar` instead of `text = str` for Label, if we want to show the content of a StringVar or IntVar.

Use `button_name = Button('page_name', text = 'str_on_button', command = lambda : funcName)`

As before, declare buttons coordinates on the page like:

`button_name.place(x= ....., y= .....,)`

For functionality of buttons, we design their purpose in a function and pass the func to the `command **kw` of button, by means of lambda.

While defining these widgets, we can add more options like `width = ....` or `highlightbackground = .....` too for more customization.

Review the concepts above, in the example below:

```
from tkinter import *

root = Tk()
root.title('My Page')
root.geometry('500x300')
root.configure(bg='cyan')

myLabel = Label(root, text='Welcome!')
myLabel.place(x=10, y=10)

myVar = StringVar()
myVar.set('Enter your age and your name:')

myLabel2 = Label(root, textvariable=myVar)
myLabel2.place(x=10, y=30)

age = IntVar()
name = StringVar()

myInp1 = Entry(root, textvariable=age)
myInp1.place(x=10, y=50)
```

```


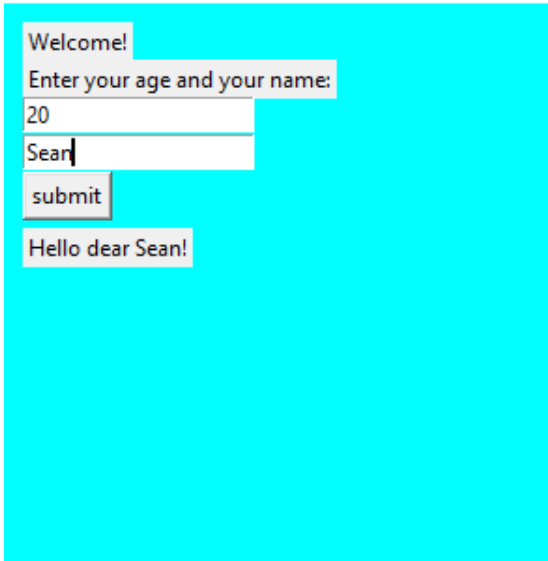
myInp2 = Entry(root)
myInp2.place(x=10, y=70)

def myFunc():
    print('this is simple print method')
    print(f'you are {age.get()} years old')
    name.set(myInp2.get())
    lbl = Label(root, text=f'Hello dear {name.get()}!')
    lbl.place(x=10, y=120)

btn = Button(root, text='submit', command=lambda:
myFunc())
btn.place(x=10, y=90)

root.mainloop()

```

Before click	After click
	
<b>Console:</b>  this is simple print method you are 20 years old	

One hint we get from last example is: as it seems, *print()* shows its argument in console, not in our window. That's why we use Labels for printing in the GUI.

Another fact is, the code after the Button or Entry *do* run even without the button clicked or entry filled; i.e., in spite of INPUT func, program doesn't wait for client's response here. But when you click, not only does the button line launch, but also all the code refreshes. E.g.:

```
from tkinter import *

page = Tk()
page.title('myPage')
page.geometry('300x200')

myVar = StringVar()

res1 = Label(page, textvariable=myVar)
res1.place(x=20, y=30)

def myFun():
    myVar.set('Hello!')

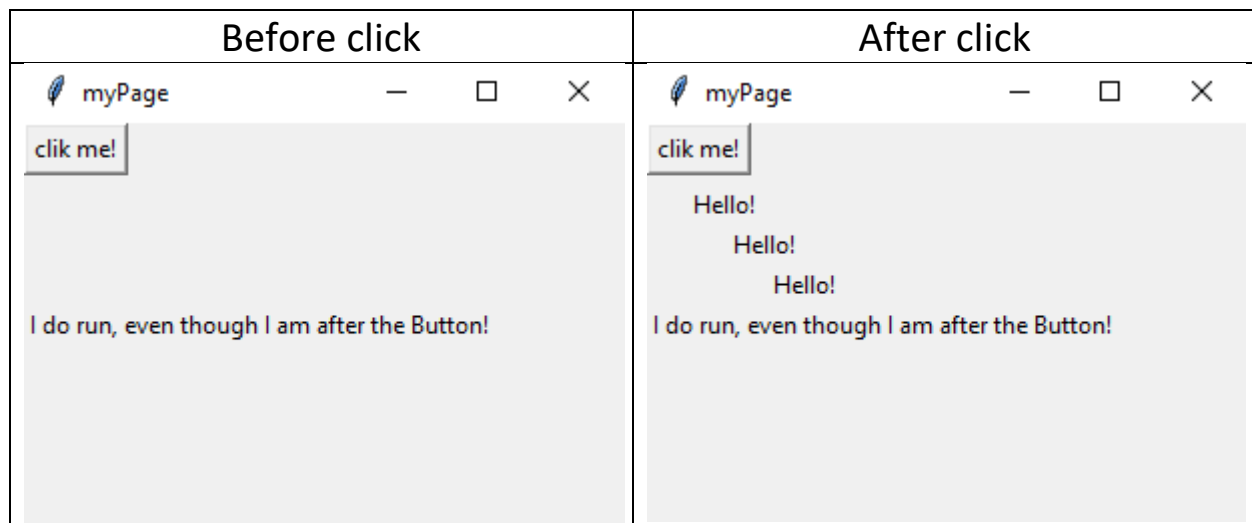
res2 = Label(page, textvariable=myVar)
res2.place(x=40, y=50)

btn = Button(page, text='klik me!', command=lambda:
myFun())
btn.place(x=0, y=0)

res3 = Label(page, text='I do run, even though I am after
the Button!')
res3.place(x=0, y=90)

res4 = Label(page, textvariable=myVar)
res4.place(x=60, y=70)

page.mainloop()
```



We can divide our window into smaller sections by framing it like:

`section_name = Frame('page_name' , width = .... , height = .... , bg = 'section_color')`

To clarify the position, you can use TOP, BOTTOM, LEFT or RIGHT like:

`section_name.pack(side=TOP)`

In every section, we can add widgets as before; but to create them, we use `section_name` instead of `page_name` as the function's master (master is the first parameter of Label, Entry, etc.; it determines on what should the object be put).

We should pack our section after introducing its widgets. We use `section_name.pack(side= ..... )` as it's been said.

We can add space between these creatures (!), horizontally and vertically, by adding `padx` and `pady` as well:

`object_name.pack(side=TOP, padx =..., pady = .....)`

In addition to `.pack()` and `.place()`, we can use `.grid(row = ..... , column = .....)` to place our objects in a neat formation on the page. Rows and columns count from zero.

Use `rowspan = .....` or `columnspan = .....` as additional specifications of your object in grid method, in order to clarify how many grid indents should your object occupy horizontally and vertically.

A list box, is a `_usually_` big space you create to show some data in it. We create them like: `list_name = Listbox('page name')`

To add data to list box, we use `list_name = insert(index, value)`

This will put the `value` on the `index` we want. You can use `END` as the last index of the list box too.

Delete data on list box like: `list_name = delete(start_index, stop_index)`

For huge data, it's usually vital to add a scrollbar next to list box to enable advancing forward and backward amongst data. This is done via `scroll_name = Scrollbar('page name')`.

Then we should connect the list box and scrollbar by `.configure()` like:

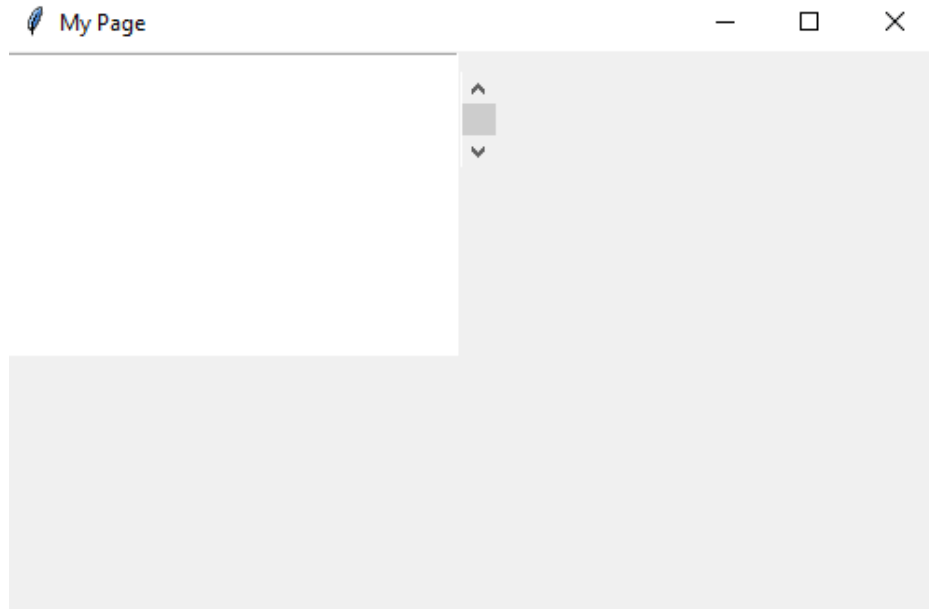
```
from tkinter import *

root = Tk()
root.title('My Page')
root.geometry('500x300')

lb = Listbox(root, width=40, height=10)
lb.grid(row=0, column=0, rowspan=5, columnspan=2)

sb = Scrollbar(root, width = 20)
sb.grid(row=0, column=4)

lb.configure(yscrollcommand=sb)
sb.configure(command=lb.yview)
root.mainloop()
```



One last example to review all we learnt about tkinter:

```
from tkinter import *

# first window
page = Tk()
page.title('welcome')
page.geometry('300x200')

# second window
root = Tk()
root.title('Logs history')
root.geometry('400x200')
root.configure(bg='blue')

# second window widgets
section = Frame(root, width=200, height=200, bg='yellow')

lb1 = Label(root, background='orange', text='logins:')
lb1.place(x=80, y=0)

lb2 = Label(section, background='orange', \
            text='logouts:')
lb2.place(x=80, y=0)
```

```

# list box 1
lbox1 = Listbox(root, width=28, height=9)
lbox1.place(x=10, y=30)

sbar1 = Scrollbar(root, width=15)
sbar1.place(x=185, y=30)

lbox1.configure(yscrollcommand=sbar1)
sbar1.configure(command=lbox1.yview)

# list box 2
lbox2 = Listbox(section, width=28, height=9)
lbox2.place(x=10, y=30)

sbar2 = Scrollbar(section, width=15)
sbar2.place(x=185, y=30)

lbox2.configure(yscrollcommand=sbar2)
sbar2.configure(command=lbox2.yview)

# packing all widgets
section.pack(side=RIGHT)

# first window widgets
lb3 = Label(page, text='username:')
lb3.grid(row=0, column=0, padx=10, pady=3)

inp = Entry(page, width=24)
inp.grid(row=0, column=1, padx=10, pady=3)

def login():
    lb4 = Label(page, text='user logged in '
                        'secussfully.')
    lb4.grid(row=5, column=1)

    lbox1.insert(END, inp.get())

```



```

def logout():
    lb4 = Label(page, text='user logged out '
                        'secussfully.')
    lb4.grid(row=5, column=1)

    lbox2.insert(END, inp.get())

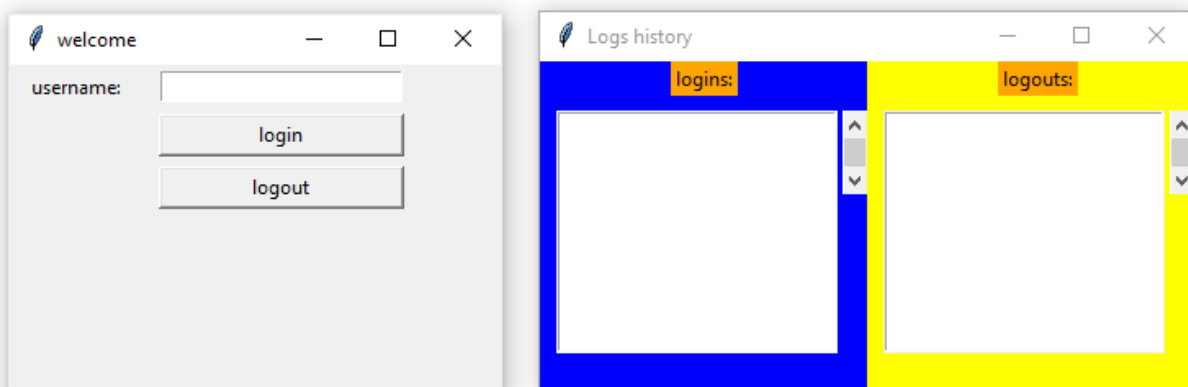
btn1 = Button(page, text='login', command=login, \
              width=20)
btn1.grid(row=1, column=1, padx=10, pady=3)

btn2 = Button(page, text='logout', command=logout, \
              width=20)
btn2.grid(row=3, column=1, padx=10, pady=3)

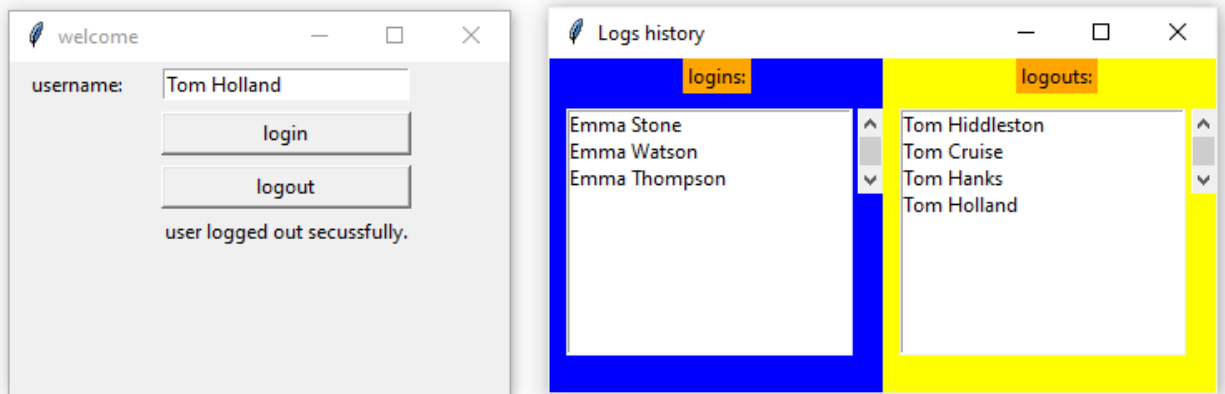
# executing
page.mainloop()
root.mainloop()

```

Raw results:



## Accomplished results:



# Appendix

## List of Python special/escape characters:

- `\n` → Newline
- `\t` → Horizontal tab
- `\r` → Carriage return
- `\b` → Backspace
- `\f` → Form feed
- `\'` → Single Quote
- `\"` → double quote
- `\\` → Backslash
- `\v` → vertical tab
- `\N` → N is the number for Unicode character
- `\NNN` → NNN is digits for Octal value
- `\xNN` → NN is a hex value; `\x` is used to denote following is a hex value.
- `\a` → bell sound, actually default chime

## Matplotlib.pyplot pre-figured styles:

(This list is accessible by *plt.styles.available*.)

```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic',  
'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale',  
'seaborn', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark',  
'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep',  
'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-  
pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-  
white', 'seaborn-whitegrid', 'tableau-colorblind10']
```

## Matplotlib.pyplot.pie() \*\*kwargs:

### Parameters:

**X:** 1D array-like

The wedge sizes.

**Explode:** array-like, default: None

If not *None*, is a len(x) array which specifies the fraction of the radius with which to offset each wedge.

**Labels:** list, default: None

A sequence of strings providing the labels for each wedge

**Colors:** array-like, default: None

A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

**Autopct:** None or str or callable, default: None

If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt % pct`. If it is a function, it will be called.

**Pctdistance:** float, default: 0.6

The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*.

**Shadow:** bool, default: False

Draw a shadow beneath the pie.

**Normalize:** None or bool, default: None

When *True*, always make a full pie by normalizing *x* so that `sum(x) == 1`. *False* makes a partial pie if `sum(x) <= 1` and raises a [ValueError](#) for `sum(x) > 1`.

When *None*, defaults to *True* if `sum(x) >= 1` and *False* if `sum(x) < 1`.

Please note that the previous default value of *None* is now deprecated, and the default will change to *True* in the next release. Please pass `normalize=False` explicitly if you want to draw a partial pie.

**Labeldistance:** float or None, default: 1.1

The radial distance at which the pie labels are drawn. If set to None, label are not drawn, but are stored for use in legend()

**Startangle:** float, default: 0 degrees

The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

**Radius:** float, default: 1

The radius of the pie.

**Counterclock:** bool, default: True

Specify fractions direction, clockwise or counterclockwise.

**Wedgeprops:** dict, default: None

Dict of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = {'linewidth': 3} to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default clip\_on=False.

**Textprops:** dict, default: None

Dict of arguments to pass to the text objects.

**Center:** (float, float), default: (0, 0)

The coordinates of the center of the chart.

**Frame:** bool, default: False

Plot Axes frame with the chart if true.

**Rotatelabels:** bool, default: False

Rotate each label to the angle of the corresponding slice if true.

# References

1. Mohammad Ordookhani, Toplearn, <https://toplearn.com/c/o2V3>
2. Amirhossein Bigdelu, Mongard, <https://www.mongard.ir/courses/python-beginner-course/>
3. Python Official Website, [Welcome to Python.org](https://www.python.org/), <https://www.python.org/>
4. [GeeksforGeeks | A computer science portal for geeks](https://www.geeksforgeeks.org/), <https://www.geeksforgeeks.org/>
5. [Stack Overflow - Where Developers Learn, Share, & Build Careers](https://stackoverflow.com/), <https://stackoverflow.com/>
6. [What is Python? Executive Summary | Python.org](https://www.python.org/doc/essays/blurb/), <https://www.python.org/doc/essays/blurb/>
7. [Stack Overflow Insights - Developer Hiring, Marketing, and User Research](https://insights.stackoverflow.com/survey/), <https://insights.stackoverflow.com/survey/>
8. [PyCharm: the Python IDE for Professional Developers by JetBrains](https://www.jetbrains.com/pycharm/), <https://www.jetbrains.com/pycharm/>
9. [Google Colab](https://colab.research.google.com/), <https://colab.research.google.com/>
10. [Write a long string on multiple lines in Python | note.nkmk.me](https://note.nkmk.me/en/python-long-string/), <https://note.nkmk.me/en/python-long-string/>
11. [Python Random Module \(w3schools.com\)](https://www.w3schools.com/PYTHON/module_random.asp), [https://www.w3schools.com/PYTHON/module\\_random.asp](https://www.w3schools.com/PYTHON/module_random.asp)
12. Sympy Official Website, [SymPy](https://www.sympy.org/en/index.html), <https://www.sympy.org/en/index.html>
13. [Solvers - SymPy 1.11 documentation](https://docs.sympy.org/latest/modules/solvers/solvers.html), <https://docs.sympy.org/latest/modules/solvers/solvers.html>
14. [Plotting - SymPy 1.11 documentation](https://docs.sympy.org/latest/modules/plotting.html), <https://docs.sympy.org/latest/modules/plotting.html>
15. Scipy Official Website, [SciPy](https://scipy.org/), <https://scipy.org/>



16. Matplotlib Official Website, [Matplotlib — Visualization with Python](https://matplotlib.org/), <https://matplotlib.org/>
17. [Text in Matplotlib Plots — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/text/text_intro.html), [https://matplotlib.org/stable/tutorials/text/text\\_intro.html](https://matplotlib.org/stable/tutorials/text/text_intro.html)
18. [Text properties and layout — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/text/text_props.html#sphx-glr-tutorials-text-text-props-py), [https://matplotlib.org/stable/tutorials/text/text\\_props.html#sphx-glr-tutorials-text-text-props-py](https://matplotlib.org/stable/tutorials/text/text_props.html#sphx-glr-tutorials-text-text-props-py)
19. [Annotations — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/text/annotations.html#sphx-glr-tutorials-text-annotations-py), <https://matplotlib.org/stable/tutorials/text/annotations.html#sphx-glr-tutorials-text-annotations-py>
20. [matplotlib.pyplot.axis — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axis.html), [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.axis.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axis.html)
21. [Linestyles — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html#linestyles), [https://matplotlib.org/stable/gallery/lines\\_bars\\_and\\_markers/linestyles.html#linestyles](https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html#linestyles)
22. <https://xkcd.com/color/rgb/>
23. [Customizing Matplotlib with style sheets and rcParams — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/introductory/customizing.html?highlight=axes.prop_cycle#a-sample-matplotlibrc-file), [https://matplotlib.org/stable/tutorials/introductory/customizing.html?highlight=axes.prop\\_cycle#a-sample-matplotlibrc-file](https://matplotlib.org/stable/tutorials/introductory/customizing.html?highlight=axes.prop_cycle#a-sample-matplotlibrc-file)
24. [Matplotlib documentation — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/index.html), <https://matplotlib.org/stable/index.html>
25. [HTML Color Codes](https://htmlcolorcodes.com/), <https://htmlcolorcodes.com/>
26. [matplotlib.markers — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/api/markers_api.html), [https://matplotlib.org/stable/api/markers\\_api.html](https://matplotlib.org/stable/api/markers_api.html)
27. [LaTeX - A document preparation system \(latex-project.org\)](https://www.latex-project.org/), <https://www.latex-project.org/>

28. [Writing mathematical expressions — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/text/mathtext.html),  
<https://matplotlib.org/stable/tutorials/text/mathtext.html>
29. [How to Create Different Subplot Sizes in Matplotlib? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-create-different-subplot-sizes-in-matplotlib/), <https://www.geeksforgeeks.org/how-to-create-different-subplot-sizes-in-matplotlib/>
30. [8 key differences between Bar graph and Histogram chart | Syncfusion](https://www.syncfusion.com/blogs/post/difference-between-bar-graph-and-histogram-chart.aspx), <https://www.syncfusion.com/blogs/post/difference-between-bar-graph-and-histogram-chart.aspx>
31. [Choosing Colormaps in Matplotlib — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/tutorials/colors/colormaps.html),  
<https://matplotlib.org/stable/tutorials/colors/colormaps.html>
32. [Plot types — Matplotlib 3.5.3 documentation](https://matplotlib.org/stable/plot_types/index.html),  
[https://matplotlib.org/stable/plot\\_types/index.html](https://matplotlib.org/stable/plot_types/index.html)
33. [printy · PyPI](https://pypi.org/project/printy/), <https://pypi.org/project/printy/>
34. [FIGlet - hosted by PLiG](http://www.figlet.org/examples.html),  
<http://www.figlet.org/examples.html>
35. [12. Virtual Environments and Packages — Python 3.10.6 documentation](https://docs.python.org/3/tutorial/venv.html#virtual-environments-and-packages),  
<https://docs.python.org/3/tutorial/venv.html#virtual-environments-and-packages>
36. [Iterables — Python Like You Mean It](https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Iterables.html#:~:text=Definition%3A,over%20in%20a%20for%2Dloop.),  
[https://www.pythonlikeyoumeanit.com/Module2\\_EssentialsOfPython/Iterables.html#:~:text=Definition%3A,over%20in%20a%20for%2Dloop.](https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Iterables.html#:~:text=Definition%3A,over%20in%20a%20for%2Dloop.)
37. [Python Iterators \(w3schools.com\)](https://www.w3schools.com/python/python_iterators.asp),  
[https://www.w3schools.com/python/python\\_iterators.asp](https://www.w3schools.com/python/python_iterators.asp)
38. [Python Special characters \(chercher.tech\)](https://chercher.tech/python-programming/python-special-characters),  
<https://chercher.tech/python-programming/python-special-characters>