Python Introduction

What is Python?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

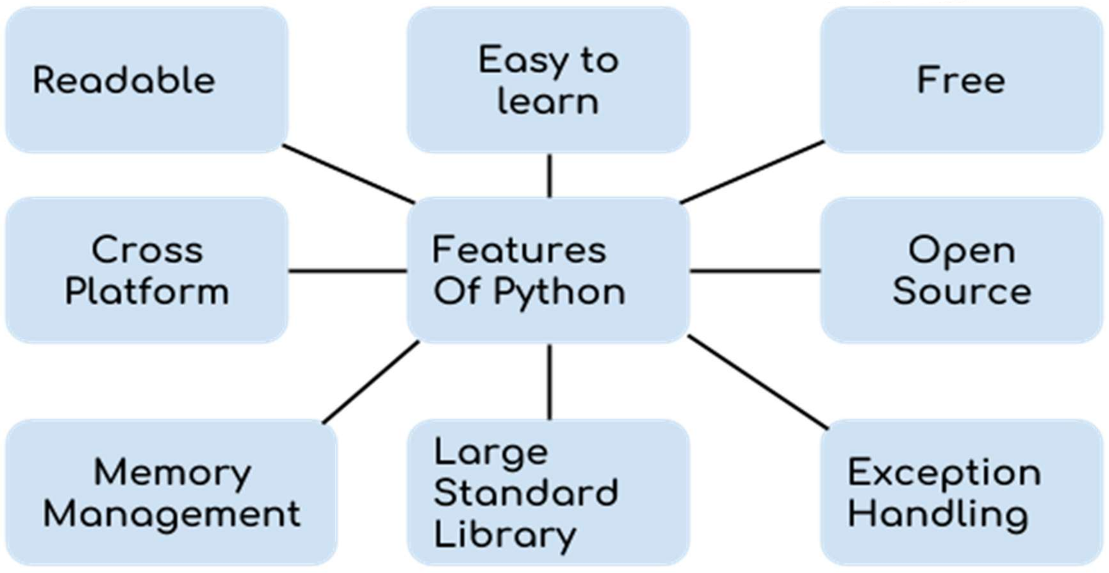
Why Python?

- Python works on different platforms (Windows, Mac, Linux, RaspberryPi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

- Python was designed for readability and has some similarities to the English language with influence from mathematics.
- Python uses newlines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions, and classes. Other programming languages often use curly brackets for this purpose.

Features of Python programming language



1. **Readable:** Python is a very readable language.

2. **Easy to Learn:** Learning python is easy as this is a expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.

3. **Cross platform:** Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.

4. **Open Source:** Python is a open source programming language.

5. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

6. **Free:** Python is free to download and use. This means you can download it for free and use it in your application. See: **Open Source Python License**. Python is an example of a FLOSS (Free/Libre Open-Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

7. **Supports exception handling:** If you are new, you may wonder what is an exception? An exception is an event that can occur during program exception and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.

8. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

Dynamic Semantic:

Python uses **dynamic semantics**, meaning that its variables are dynamic **objects**.
In C language if we want to use a variable ***x***, we need to declare it. And it will take data of data type same as it has been declared. i.e. we cannot store a string value in a variable which is declared as *int*. Whereas in Python the variable will take same data type as the data stored in it. i.e. if we store a string in a variable ***x*** then it will be of string type and immediately if we store an integer value then it will take data type as integer.

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

1. Single Line Comments :

Comments start with a '#'

Example :

```
#This is a comment
print("Hello, World!")
```

2. Multi Line Comments :

- Python does not really have a syntax for multi-line comments.
- To add a multiline comment, you could insert a '#' for each line:

Example :

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes-3 times single/double quotes) in your code, and place your comment inside it:

Example :

```
"""
This is a comment
```

```
written in
more than just one line
"""

‘’’
This is another multi-line comment
written in
more than just one line
‘’’
print("Hello, World!")
```

Variables:

- Variables are containers for storing data values.

Creating Variables

- Python has no command for declaring a variable.
- A variable is created the moment you assign a value to it for the first time.

  Example:

```
x = 5
y = "John"
print(x)
print(y)
```

- Variables do not need to be declared with any particular *type* and can even change *type* after they have been set.

  Example:

```
x = 4                # x will be of type int
x = "VNSGU"          # x will now be of type str
print(x)
```

Get the Data Type of a Variable:

- You can get the data type of a variable with the type() function.

  Example

```
x = 5
y = "Surat"
print(type(x))
print(type(y))
```

Single or Double Quotes?

- String variables can be declared either by using single or double quotes:

  Example:

  x = "DCS"
  # is the same as
  x = 'DCS'

Case-Sensitive:

Variable names are case-sensitive.

Example

The following code will create two different variables:

a = 4
A = "DCS"
#A will not overwrite a

Python - Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:
- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Multi Words Variable Names

- Variable names with more than one word can be difficult to read.
- There are several techniques you can use to make them more readable:

Camel Case: Each word, except the first, starts with a capital letter:

myVariableName = "Rajesh"

Pascal Case: Each word starts with a capital letter:

MyVariableName = "Rajesh"

Snake Case: Each word is separated by an underscore character:

my_variable_name = "Rajesh"

<u>Python Variables - Assign Multiple Values</u>

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpack a Collection

If you have a collection of values in a list, tuple etc.

Python allows you extract the values into variables. This is called *unpacking*.

Example

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Namespace

A ***namespace*** is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as *abs()*, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function *maximize* without confusion — users of the modules must prefix it with the module name.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace. (The built-in names actually also live in a module; this is called builtins.)

The local namespace for a function is created when the function is called and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the nonlocal statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global or nonlocal statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal statement indicates that particular variables live in an enclosing scope and should be rebound there.

Python Lambda

A lambda function is a small anonymous function.
A lambda function can take any number of arguments but can only have one expression.

```
lambda arguments : expression
```

The expression is executed, and the result is returned:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

```
x = lambda a, b : a * b
print(x(5, 6))

x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
        return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```

Examples:

```
def evaluate(f, x, y):    # f is name of a function
        print(f, type(f))
        print(x, y)
        print(10*'-')
        return f(x, y)

print(evaluate(lambda x, y: 3*x + y, 10, 2))
evaluate(lambda x, y: print(x, y), 10, 2)
print(evaluate(lambda x, y: 10 if x == y else 2, 5, 5))
print(evaluate(lambda x, y: 10 if x == y else 2, 5, 3))
```

# Lambda function/expression

```
line=50*'='

'''
def f(x):
        return x*5
'''

f=lambda x: x*5

for i in range(1,5):
        print(f'f({i}): {f(i)}')

sum = lambda a, b: a+b
print('sum(10, 12):', sum(10, 12))
print('sum(20, 45):', sum(20, 45))

def mult(n):
        return lambda x: x*n

multi5=mult(5)
multi3=mult(3)

print(multi5(10), multi3(10))

for i in range(1,11):
        print("%2d X 5 = %2d"% (i, multi5(i)))

for i in range(1,11):
        print("%2d X 3 = %2d"% (i, multi3(i)))
```

```
pairs=[(1, 'One'), (4, 'Four'), (2, 'Two'), (6, 'Six'), (3, 'Three'), (5, 'Five')]
print(pairs)
print('Ascending Order on 1st Value'.center(75,'*'))
pairs.sort()
print(pairs)
print('Ascending Order on 2nd Value'.center(75,'*'))
pairs.sort(key=lambda pair:pair[1])
print(pairs)
pairs=[(1, 'O', 'S'), (4, 'F', 'E'), (2, 'T', 'B'), (6, 'S', 'A'), (3, 'T', 'C')]
print('Ascending Order on 3rd Value'.center(75,'*'))
pairs.sort(key=lambda pair:pair[2])
print(pairs)
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)

letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']
vowels = list(filter(lambda x: x in ['a', 'e', 'i', 'o', 'u'], letters))
print(vowels)

my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)

num1 = [4, 5, 6]
num2 = [5, 6, 7]

result = map(lambda n1, n2: n1+n2, num1, num2)
print(list(result))

##Example to explain map function

def calcSquare(n):
        return n*n

numbers = (1, 2, 3, 4)
result = map(calcSquare, numbers)
print(result)

# converting map object to list
numbersSquare = list(result)
print(numbersSquare)
```

File Handling

Mode	Action
r	Opens a Text file for reading only. The File pointer is placed at the beginning of the file. This is the default mode. Error if the file does not exist.
rb	same as above for Binary files
r+	Opens a Text file for both reading and writing. The File pointer is placed at the beginning of the file. Error if the file does not exist.
rb+	same as above for Binary files
w	Opens a Text file for writing only. Overwrites the file if it exists & creates if it doesn't exist
wb	same as above for Binary files
w+	Opens a Text file for both writing and reading. Overwrites the file if it exists & creates if it doesn't exist
wb+	same as above for Binary files
a	Opens a Text file for appending. The File pointer is placed at the end of the file if it exists. That is, it is in append mode. If the file does not exist, it creates a new file for writing.
ab	same as above for Binary files
a+	Opens a Text file for both appending and reading. The File pointer is placed at the end of the file if it exists. That is, it is in append mode. If the file does not exist, it creates a new file for reading & writing.
ab+	same as above for Binary files

File Object Attributes:

file.closed	Returns true if file is closed, false otherwise
file.mode	Returns access mode with which the file was opened
file.name	Returns name of the file

open() returns a file object and is most commonly used with two arguments:

filehandler=open(filename, [mode]). **(default value of mode is *'r'*)**

Reading a file:

If you know the ***file is relatively small compared to the size of your main memory***, you can ***read the whole file into one string using the read() method*** on the file handle. If the file is too large to fit in main memory, you should write your program to read the file in chunks using a for or while loop.

To read a file's contents, call ***f.read([size])***, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). ***size* is an optional numeric argument**. ***When size is omitted or negative, the entire contents of the file will be read and returned***; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most size bytes are read and returned. ***If the end of the file has been reached, f.read() will return an empty string ('').***

```
line=fhandle.read([size])
```

When the file is read in this manner, ***all the characters including all of the lines and newline characters are one big string in the variable line***. It is a good idea to store the output of ***read*** as a variable because each call to read exhausts the resource:

It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
with open(fname) as f:          #here f will be the file handler
        read_data=f.read()
print(read_data)
```

alternative of

```
f=open(fname)
read_data=f.read()
print(read_data)
f.close()
```

```
with open(fname, 'w') as fhand:
        fhand.write('File creation\n')
        while True:
                txt=input()
                if txt.upper() == 'END': break
                fhand.write(txt+'\n')
```

----

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; ***if f.readline() returns an empty string, the end of the file has been reached***, while a blank line is represented by '\n', a string containing only a single newline.

```
while True:
        line=fhand.readline().strip()   # strip removes leading and trailing whitespace
                                          characters
        if len(line)==0: break
        print(line)
```

If you want to **read all the lines of a file in a list** you can also use ***list(f)*** or ***f.readlines()***.

Both the ***read()*** and ***readlines()*** method read a whole file at once. Obviously, for small files this is acceptable, but for long files you might not have enough memory to store the file contents efficiently. In such circumstances (or when you do not know the file size), you should read a file line by line with the ***readline()*** method.

Once you've opened up a file, you'll want to read or write to the file. First, let's cover reading a file. There are multiple methods that can be called on a file object to help you out:

Method	What It Does
read(size=-1)	This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.
readline(size=-1)	This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.
readlines()	This reads the remaining lines from the file object and returns them as a list.

readlines() method returns the contents of the entire file as a list of strings, where each item in the list represents one line of the file.

read() reads the entire file into a single string

```
fhandle=open(fname) (OR fhandle=open(fname, 'r'))

for line in fhandle:
        print(line)
```

When the file is read using a ***for loop*** in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and ***includes the newline as the last character in the line variable for each iteration of the for loop***.

Writing using *write()*

f.write(string) writes the contents of string to the file, ***returning the number of characters written***.

f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use ***f.seek(offset, whence)***.

- The position is computed from ***adding offset to a reference point***; the ***reference point is*** selected by the ***whence*** argument.

- A whence value of:
    - ***0*** measures from the ***beginning of the file***,
    - ***1*** uses the ***current file position***, and
    - ***2*** uses the ***end of the file*** as the reference point.

- ***whence*** can be omitted and ***defaults to 0***, using the beginning of the file as the reference point.

In ***text files*** (those opened without a ***b*** in the mode string), ***only seeks relative to the beginning of the file are allowed*** (the exception being ***seeking to the very file end with seek(0, 2)***) and the only valid offset value is zero. Any other offset value produces undefined behaviour.

Delete a File:

To delete a file, you must ***import*** the ***OS*** module, and run its ***os.remove()*** function:

```
import os
os.remove(fname)
```

Check if file exists, *then* delete it:

```
import os
if os.path.exists(fname):
        os.remove(fname)
else:
        print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the ***os.rmdir()*** method:

```
import os
os.rmdir(foldername)
```

Note: You can only remove *empty* folders.

Python Modules

What is a Module?

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension ***.py***:

Example

Save this code in a file named ***mymodule.py***

```
def greeting(name):
        print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the ***import*** statement:

Example

Import the module named ***mymodule***, and call the greeting function:

```
import mymodule

mymodule.greeting("Ravi")
```

Note: When using a function from a module, use the syntax: ***module_name.function_name***.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file ***mymodule.py***

```
person1 = {"name" : "Ajay", "age" : 36, "country" : "India"}
```

Example

Import the module named ***mymodule***, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
```

```
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension ***.py***

Re-naming a Module

You can create an alias when you import a module, by using the ***as*** keyword:

Example

Create an alias for ***mymodule*** called ***mylib***:

```
import mymodule as mylib

a = mylib.person1["age"]

print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

To display all the available modules, use the following command in the Python Console:

```
help('modules')
```

Enter any module name to get more help. Or, type "modules spam" to search
for modules whose name or summary contain the string "spam".

Example

Import and use the ***platform*** module:

```
import platform

x = platform.system()

print(x)
```

Using the *dir()* Function

There is a built-in function to list all the function names (or variable names) in a module. The ***dir()*** function:

Example

List all the defined names belonging to the platform module:

```
import platform
```

```
x = dir(platform)

print(x)
```

Note: The ***dir()*** function can be used on ***all*** modules, also the ones you create yourself.

Import from Module

You can choose to import only parts from a module, by using the ***from*** keyword.

from modulename import functionnames
from modulename import variablenames

Example

The module named ***mymodule*** has one function and one dictionary:

```
def greeting(name):
        print("Hello, " + name)

person1 = {"name" : "Ajay", "age" : 36, "country" : "India"}
```

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the from keyword, do not use the module name when referring to elements in the module. Example: ***person1["age"]***, **not** ~~mymodule.person1["age"]~~

Python - OS Module

It is possible to automatically **perform many operating system tasks**. The OS module in Python provides functions for **creating** and **removing** a **directory** (folder), **fetching its contents**, **changing**, and **identifying the current directory**, etc.

You first need to import the ***os*** module to interact with the underlying operating system. So, import it using the ***import os*** statement before using its functions.

import os

Function	**Use**
os.getcwd()	returns the current working directory
os.mkdir(dir_name)	create a new directory
os.chdir(dir_name)	change current working directory change the current working directory to a drive ***os.chdir('C:\\')***

	set the current directory to the parent directory ***os.chdir('..')***
os.rmdir(dir_name)	removes the specified directory either with an absolute or relative path. **Note that, for a directory to be removed, it should be empty.**
os.listdir(dir_name)	returns the list of all files and directories in the specified directory

Python - sys Module

The sys module provides functions and variables used to **manipulate different parts of the Python runtime environment**. You will learn some of the important features of this module here.

import sys

Function	**Use**
sys.argv(index)	• It returns a list of command line arguments passed to a Python script. • The item at index 0 in this list is always the name of the script. • The rest of the arguments are stored at the subsequent indices.
sys.exit	• This causes the script to exit back to either the Python console or the command prompt. • This is generally used to safely exit from the program in case of generation of an exception.
sys.maximize	Returns the largest integer a variable can take.
sys.path	This is an environment variable that is a search path for all Python modules.
sys.version	This attribute displays a string containing the version number of the current Python interpreter.

Python - Math Module

Some of the most popular mathematical functions are defined in the math module. These include **trigonometric functions**, **representation functions**, **logarithmic functions**, **angle conversion functions**, etc. In addition, **two mathematical constants** are also defined in this module.

- **Pi** is a well-known mathematical constant and its value is **3.141592653589793**.
- **e** is another well-known mathematical constant. It is called **Euler's number** and it is a base of the natural logarithm. Its value is **2.718281828459045**.

import math

Function	**Use**
math.degrees(radian)	Converts angle from radians to degrees.
math.radians(degree)	Converts angle from degrees to radians.

math.sin(radians)
math.cos(radians)
math.tan(radians)
	Calculate trigonometric ratios **sin**, **cos** and **tan** for a given angle in radians.

math.log(number)	Natural logarithm of a given number. The natural logarithm is calculated to the **base e**.
math.log10(number)	Base 10 logarithm of a given number. It is called standard logarithm.
math.exp(number) ***math.e**number***	Returns a float number after raising e to the power of the given number. In other words, **exp(x)** gives **e**x**.
math.pow(float, float)	This method receives two float arguments, raises the first to the second and returns the result. In other words, **pow(5, 4)** is equivalent to **5**4**.
math.sqrt(number)	Returns the square root of the number.
math.ceil(number)	It approximates the given number to the smallest integer, greater than or equal to the given floating-point number. **(representation function)**
math.floor(number)	It returns the largest integer less than or equal to the given number. **(representation function)**

Python - Statistics Module

The statistics module provides functions to mathematical statistics of numeric data. The following popular statistical functions are defined in this module.

import statistics

Function	**Use**
statistics.mean(list)	It calculates the **arithmetic mean** of the **numbers in a list**.
statistics.median(list)	It calculates the **middle value** of the **numeric data in a list**.
statistics.mode(list)	It returns the **most common data point** in the **list**.
statistics.stdev(list)	It calculates the **standard deviation on a given sample** in the form of a **list**.

Python - Collections Module

import collections

The collections module **provides alternatives to built-in container data types** such as list, tuple and dict.

namedtuple()

The ***namedtuple()*** function returns a tuple-like object with named fields. These field attributes are accessible by lookup as well as by index.

General usage of this function is:

collections.namedtuple(type_name, field-list)

The following statement declares a student class having name, age and marks as fields.

Example: Declare a Named Tuple

```
>>> import collections
>>> student = collections.namedtuple('student', [name, age, marks])
```

To create a new object of this ***namedtuple***, do the following:

Example: Create Object of Named Tuple

```
>>> s1 = student(‘Rahul’, 21, 98)
```

The values of the field can be accessible by attribute lookup:

Example: Access Named Tuple

```
>>> s1.name
‘Rahul'
```

Or by index:

Example: Access Named Tuple

```
>>> s1[0]
'Rahul'
```

OrderedDict()

The ***OrderedDict()*** function is similar to a normal dictionary object in Python. However, it remembers the order of the keys in which they were first inserted.

Example: Ordered Dictionary

```
import collections

d1 = collections.OrderedDict()
d1['A'] = 65
d1['C'] = 67
d1['B'] = 66
d1['D'] = 68

for k,v in d1.items():
        print (k,v)
```

Output

```
A 65
C 67
B 66
D 68
```

Upon traversing the dictionary, pairs will appear in the order of their insertion.

deque()

A ***deque*** object support **appends** and **pops** from **either ends of a list**. It is **more memory efficient than a normal list object**. In a normal list object, the removal of any item causes all items to the right to be shifted towards left by one index. Hence, **it is very slow**.

Example: ***Deque***

```
>>> q=collections.deque([10,20,30,40])
>>> q.appendleft(0)
>>> q
deque([0, 10, 20, 30, 40])
>>> q.append(50)
>>> q
deque([0, 10, 20, 30, 40, 50])
>>>q.pop()
50
>>> q
deque([0, 10, 20, 30, 40])
>>> q.popleft()
0
>>> q
deque([10, 20, 30, 40])
```

Python - Random Module

The **random** module is a built-in module to generate the pseudo-random variables. It can be used perform some action randomly such as to get a random number, selecting a random elements from a list, shuffle elements randomly, etc.

```
import random
```

Generate **Random Floats**

The ***random.random()*** method returns a random float number **between 0.0 to 1.0**. The function doesn't need any arguments.

random.random()

Generate **Random Integers**

The ***random.randint()*** method returns a **random integer between the specified integers**.

random.randint(x, y)

Generate **Random Numbers within Range**

The ***random.randrange()*** method returns a randomly selected **element from the range created by the start, stop and step arguments**. The value of **start is 0 by default**. Similarly, the value of **step is 1 by default**.

Select **Random Elements**

The ***random.choice()*** method returns **a randomly selected element from a non-empty sequence**. An **empty sequence** as argument **raises** an ***IndexError***.

Example:

```
>>> import random
>>> random.choice('computer')
't'
>>> random.choice([12,23,45,67,65,43])
45
>>> random.choice((12,23,45,67,65,43))
67
```

Shuffle Elements Randomly

The ***random.shuffle()*** method randomly reorders the elements in a ***list***.

Example:

```
>>> numbers=[12,23,45,67,65,43]
>>> random.shuffle(numbers)
>>> numbers
[23, 12, 43, 65, 67, 45]
>>> random.shuffle(numbers)
>>> numbers
[23, 43, 65, 45, 12, 67]
```

Python Datetime

Python Dates

A date in Python is not a data type of its own, but we can import a module named ***datetime*** to work with dates as date objects.

Example

Import the ***datetime*** module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

- The date contains ***year***, ***month***, ***day***, ***hour***, ***minute***, ***second***, and ***microsecond***.
- The ***datetime*** module has many methods to return information about the date object.

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the ***datetime()*** **class (constructor)** of the ***datetime*** module.

The ***datetime()*** **class** requires three parameters to create a date: ***year***, ***month***, ***day***.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2021, 4, 10)
print(x)
```

The ***datetime()*** class also takes parameters for time and timezone (***hour***, ***minute***, ***second***, ***microsecond***, ***tzone***), but they are optional, and has a **default** value of ***0***, (**None for timezone**).

The *strftime()* Method

The ***datetime*** object has a method for formatting date objects into readable strings.

The method is called ***strftime()***, and takes one parameter, ***format***, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2021, 4, 10)
print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	**Description**	**Example**
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	01

Python Module Attributes: name, doc, file, dict

Python module has its attributes that describes it. Attributes perform some tasks or contain some information about the module. Some of the important attributes are explained below:

Attribute	**Use**
__name__	returns the name of the module. By default, the name of the file (excluding the extension ***.py***) is the value of ***__name__*** attribute.
__doc__	denotes the documentation string (docstring) line written in a module code. ***"""This is docstring of test module"""*** *def SayHello(name):* *print ("Hi {}! How are you?".format(name))* *return*
__file__	optional attribute which holds the name and path of the module file from which it is loaded. *>>> import io* *>>> io.__file__* *'C:\\python37\\lib\\io.py'*
__dict__	return a dictionary object of module attributes, functions and other definitions and their respective values.

dir() is a built-in function that also returns the list of all attributes and functions in a module.

Python Classes/Objects

Object Oriented Programming (OOP)

- Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.
- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- Almost everything in Python is an ***object***, with its ***properties (attributes/data)*** and ***methods***.
- A ***Class*** is like an object constructor, or a "**blueprint**" for creating objects.
- We can think of ***class*** as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
- Classes provide a means of bundling data and functionality together.
- As many houses can be made from a house's blueprint, we can create many objects from a class.
- An ***object*** is also called ***an instance of a class*** and the process of creating this object is called ***instantiation***.
- Python classes provide all the standard features of Object-Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes participate of the dynamic nature of Python: they are created at runtime and can be modified further after creation.

An object has two characteristics:

- attributes
- behaviour

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, colour as attributes
- singing, dancing as behaviour

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Defining a Class in Python

- Like function definitions begin with the ***def*** keyword in Python, class definitions begin with a ***class*** keyword.
- The first string inside the class is called ***docstring*** and has a brief description about the class. Although not mandatory, this is **highly recommended**.

Here is a simple class definition.

```
class MyNewClass:
'''This is a docstring. I have created a new class'''
        pass
```

- A ***class*** creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
- There are also special attributes in it that begins with double underscores **__**. For example, ***__doc__*** gives us the **docstring** of that class.
- As soon as we define a class, a new class object is created with the same name.
- Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods (defined by its class) for modifying its state.
- This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Example

```
class Person:
        "This is a person class"
        age = 10

        def greet(self):
                print('Hello')

# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes. It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> harry = Person()
```

This **will create a new object instance named harry**. We can access the attributes of objects using the object name prefix.
Attributes may be **data** or **method**. Methods of an object are corresponding functions of that class. This means to say, since ***Person.greet*** is a **function object (attribute of class)**, ***Person.greet*** will be a method object.

Example

```
class Person:
   "This is a person class"
   message='How are you?'   # class variable shared by all instances

   def __init__(self, name='World'):
      self.name=name            # instance variable unique to each instance

   def greet(self):
      print(f'Hello {self.name}!')
      print(f'{self.message}\n')

# create a new object of Person class
x=Person()
y=Person('Harry')
z=Person('Potter')

print(x.name)          # the attribute "name" unique to x
print(x.message)       # the attribute "message" shared by all Persons
x.greet()

# Output
World
How are you?
Hello World! How are you?

print(y.name)
print(y.message)
y.greet()

# Output
Harry
How are you?
Hello Harry! How are you?

print(z.name)
print(z.message)
```

```
z.greet()

# Output
Potter
How are you?
Hello Potter! How are you?

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(x.greet)

# Calling object's greet() method
# Output: Hello
x.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

- whenever an object calls its method, the object itself is passed as the first argument. So, ***x.greet()*** translates into ***Person.greet(x)***.
- calling a method with a list of ***n*** arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.
- the first argument of the function in class must be the object itself. This is conventionally called ***self***. It can be named otherwise but it is highly recommended to follow the convention.
- here ***message*** is a **class variable shared** by all the instances of the class ***Person***.
- whereas, ***name*** is an ***instance variable*** **unique** to each instances of the class ***Person***.

- shared data can have possibly surprising effects with involving **mutable** objects such as lists and dictionaries. For example, the ***tricks*** list in the following code **should not be used as a class variable** because just a **single list would be shared by all *Dog* instances**:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
>>> class Warehouse:
            purpose = 'storage'
            region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
```

```
class C:
  f = f1

  def g(self):
    return 'hello world'

  h = g
```

Now ***f***, ***g*** and ***h*** are all attributes of ***class C*** that refer to function objects, and consequently they are all methods of instances of ***C*** — ***h*** being exactly equivalent to ***g***. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the ***self*** argument:

```
class Bag:
  def __init__(self):
    self.data = []

  def add(self, x):
    self.data.append(x)

  def addtwice(self, x):
    self.add(x)
    self.add(x)
```

Constructors in Python

- Class functions that begin with double underscore __ are called special functions as they have special meaning.
- Of one particular interest is the ***__init__()*** function. This special function gets called whenever a new object of that class is instantiated.
- This type of function is also called **constructors** in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Example

```
class ComplexNumber:
   def __init__(self, r=0, i=0):
         self.real = r
         self.imag = i

   def get_data(self):
         print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method
# Output: 2+3j
```

```
num1.get_data()

# Create another ComplexNumber object and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

# Output: (5, 0, 10)
print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

Output

```
2+3j
(5, 0, 10)
Traceback (most recent call last):
  File "<string>", line 27, in <module>
    print(num1.attr)
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

- In the above example, we defined a new class to represent complex numbers. It has two functions, ***__init__()*** to initialize the variables (defaults to zero) and ***get_data()*** to display the number properly.
- An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute ***attr*** for object ***num2*** and read it as well. But this does not create that attribute for object ***num1***.

Deleting Attributes and Objects

Any attribute of an object can be **deleted** anytime, using the ***del*** statement. Try the following on the Python shell to see the output.

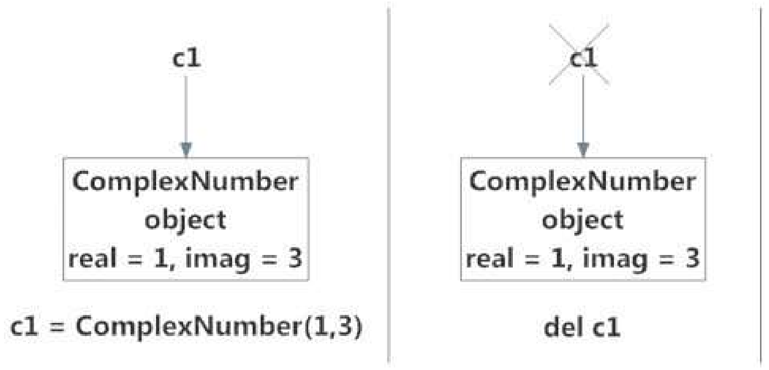
del obj_name.attr_name
del obj_name
del class_name.method

```
>>> num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

- We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

- Actually, it is more complicated than that. When we do ***c1 = ComplexNumber(1,3)***, a new instance object is created in memory and the name ***c1*** binds with it.
- On the command ***del c1***, this binding is removed and the name ***c1*** is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
- This automatic destruction of unreferenced objects in Python is also called ***garbage collection***.



c1
ComplexNumber object real = 1, imag = 3
c1 = ComplexNumber(1,3)
c1
ComplexNumber object real = 1, imag = 3
del c1

Deleting objects in Python removes the name binding

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colours, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be:

```
class Parrot:
    pass
```

Here, we use the ***class*** keyword to define an empty class ***Parrot***. **From class, we construct instances**. An **instance is a specific object created from a particular class**.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, ***obj*** is an object of ***class Parrot***.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name ***Parrot***. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the ***__init__*** method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the ***Parrot*** class. Here, *blu* and *woo* are references (value) to our new objects.

We can access the class attribute using ***__class__.species***. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using ***blu.name*** and ***blu.age***. However, instance attributes are different for every instance of a class.

Create a Class

To create a class, use the keyword ***class***:

Example

Create a **class** named ***MyClass***, with a **property** named ***x***:

```
class MyClass:
        x = 5
```

Create Object
Now we can use the class named *MyClass* to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The pass Statement

Class definitions cannot be empty, but if, for some reason, you have a ***class*** definition with no content, put in the ***pass*** statement to avoid getting an error.

Example

```
class Person:
        pass
```

Python Inheritance

- Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more. In this tutorial, you will learn to use inheritance in Python.
- It refers to defining a new ***class*** with little or no modification to an existing class. The new class is called ***derived*** (***or child***) ***class*** and the one from which it inherits is called the ***base*** (***or parent***) ***class***.

```
class DerivedClassName(BaseClassName):
   <statement-1>
   .
   .
   .
   <statement-N>
```

The name ***BaseClassName*** must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call ***BaseClassName.methodname(self, arguments)***. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as ***BaseClassName*** in the global scope.)

Python has two built-in functions that work with inheritance:

- Use ***isinstance()*** to check an instance's type: ***isinstance(obj, int)*** will be ***True*** only if ***obj.__class__*** is ***int*** or some class derived from ***int***.
- Use ***issubclass()*** to check class inheritance: ***issubclass(bool, int)*** is ***True*** since ***bool*** is a subclass of ***int***. However, ***issubclass(float, int)*** is ***False*** since ***float*** is not a subclass of ***int***.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
        def __init__(self, no_of_sides):
                self.n = no_of_sides
                self.sides = [0 for i in range(no_of_sides)]
        def inputSides(self):
            self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in
            range(self.n)]

        def dispSides(self):
            for i in range(self.n):
                print("Side",i+1,"is",self.sides[i])
```

- This ***class*** has data attributes to store the number of sides n and magnitude of each side as a list called ***sides***.
- The ***inputSides()*** method takes in the magnitude of each side and ***dispSides()*** displays these side lengths.
- A triangle is a polygon with 3 sides. So, we can create a ***class*** called ***Triangle*** which ***inherits*** from ***Polygon***. This makes all the attributes of ***Polygon class*** available to the ***Triangle class***.
- We don't need to define them again (code reusability). ***Triangle*** can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class ***Triangle*** has a new method ***findArea()*** to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

- We can see that even though we did not define methods like ***inputSides()*** or ***dispSides()*** for class ***Triangle*** separately, we were able to use them.

- If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

- In the above example, notice that ***__init__()*** method was defined in both classes, ***Triangle*** as well ***Polygon***. When this happens, **the method in the derived class overrides that in the base class**. This is to say, ***__init__()*** in ***Triangle*** gets preference over the ***__init__*** in ***Polygon***.?>
- Generally, when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling ***Polygon.__init__()*** from ***__init__()*** in ***Triangle***).
- A better option would be to use the built-in function ***super()***. So, ***super().__init__(3)*** is equivalent to ***Polygon.__init__(self,3)*** and is preferred. The super() function in Python is explained in detail later.
- Two built-in functions ***isinstance()*** and ***issubclass()*** are used to **check inheritances**.
- The function ***isinstance()*** returns ***True*** if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

Python Multiple Inheritance

- A **class can be derived from more than one base class** in Python, similar to C++. This is called **multiple inheritance**.
- In multiple inheritance, the **features of all the base classes are inherited into the derived class**.

The syntax for multiple inheritance is similar to single inheritance.

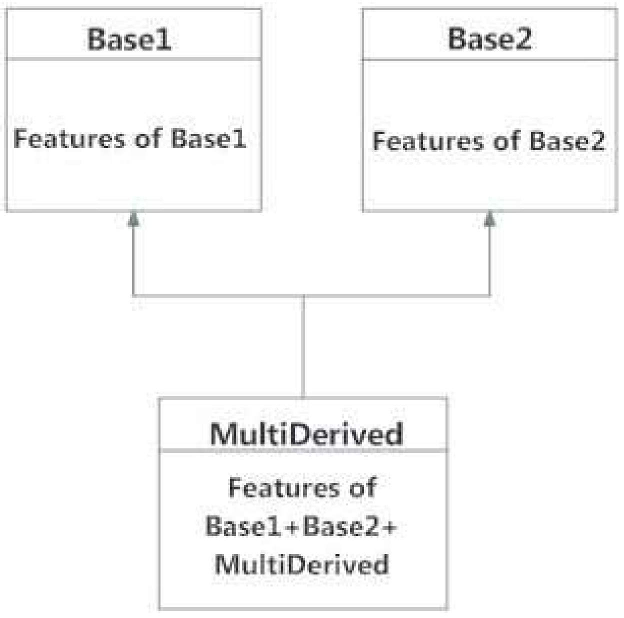
```
class DerivedClassName(Base1, Base2, Base3):
  <statement-1>
  .
  <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in ***DerivedClassName***, it is searched for in ***Base1***, then (recursively) in the base classes of ***Base1***, and if it was not found there, it was searched for in ***Base2***, and so on.

Example

```
class Base1:
  pass

class Base2:
  pass

class MultiDerived(Base1, Base2):
  pass
```

Here, the MultiDerived class is derived from Base1 and Base2 classes.



Multiple Inheritance in Python

The ***MultiDerived*** class inherits from both ***Base1*** and ***Base2*** classes.

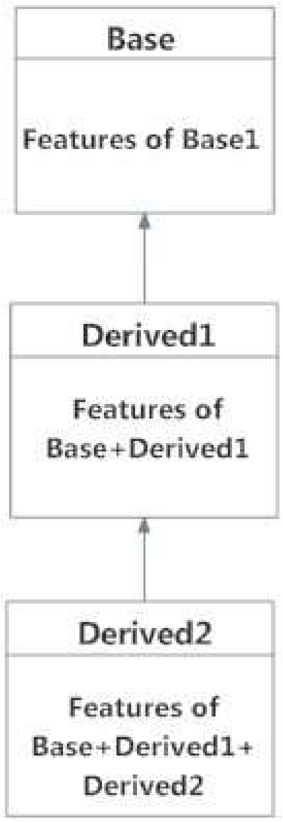
Python <u>Multilevel</u> Inheritance

- We can also **inherit from a derived class**. This is called **multilevel inheritance**. It can be of **any depth** in Python.
- In multilevel inheritance, **features of the base class and the derived class are inherited into the new derived class**.

An example with corresponding visualization is given below.

```
class Base:
   pass

class Derived1(Base):
   pass

class Derived2(Derived1):
   pass
```

Here, the ***Derived1* class** is derived from the Base class, and the ***Derived2* class** is derived from the ***Derived1* class**.


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Multilevel Inheritance in Python

Method Resolution Order in Python (MRO)

- Every class in Python is derived from the object class.
- So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.

- So, in the above example of ***MultiDerived* class** the search order is **[*MultiDerived*, Base1, Base2, object]**. This order is also called **linearization** of ***MultiDerived* class** and the set of rules used to find this order is called **Method Resolution Order (MRO)**.
- MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes.
- MRO of a class can be viewed as the ***__mro__* attribute** or the ***mro()* method**. The **former returns a tuple** while the **latter returns a list**.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

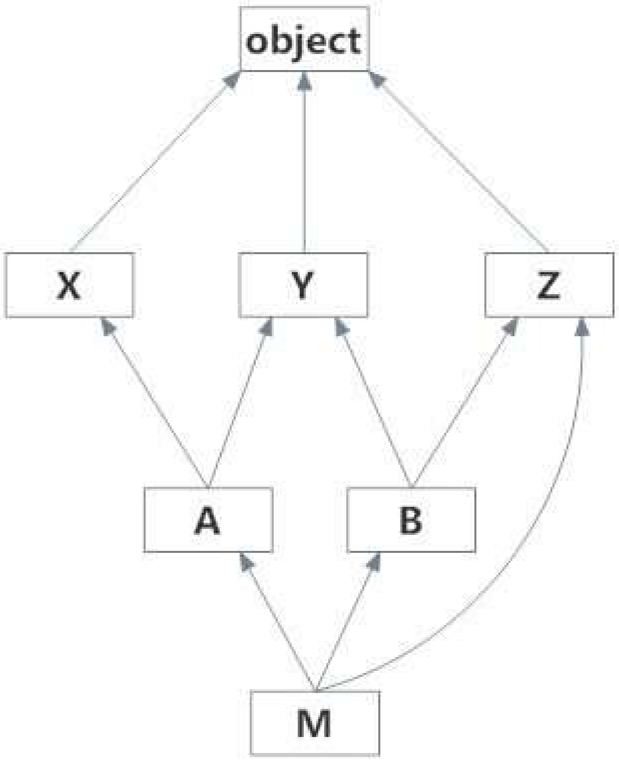
Here is a little more complex multiple inheritance example and its visualization along with the MRO.

```
# Demonstration of MRO

class X:
   pass

class Y:
   pass

class Z:
   pass

class A(X, Y):
   pass
```

```
class B(Y, Z):
    pass

class M(B, A, Z):
    pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
#  <class 'object'>]

print(M.mro())
```

Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class
'__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```



Visualizing Multiple Inheritance in Python

To know the actual algorithm on how MRO is calculated, visit Discussion on MRO.

Python super()

The ***super()*** built-in returns a **proxy object** (**temporary object of the superclass i.e. base class**) **that allows us to access methods of the base class**.

In Python, ***super()*** has two major use cases:

- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

Example 1: *super()* with Single Inheritance

In the case of **single inheritance**, it **allows us to refer base class by *super()***.

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog') # Mammal.__init__('Dog')

d1 = Dog()
```

Output

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called the __init__() method of the Mammal class (from the Dog class) using code
super().__init__('Dog')
instead of
Mammal.__init__(self, 'Dog')

Since we do not need to specify the name of the base class when we call its members, we can easily change the base class name (if we need to).

```
# changing base class to CanidaeFamily
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # no need to change this
    super().__init__('Dog')
```

- The ***super()*** built-in returns a proxy object, a substitute object that can **call methods of the base class via delegation**. This is called **indirection (ability to reference base object with *super()*)**
- Since the indirection is computed at the runtime, we can use different base classes at different times (if we need to).

Example 2: super() with Multiple Inheritance

```
class Animal:
  def __init__(self, Animal):
    print(Animal, 'is an animal.');

class Mammal(Animal):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')
    super().__init__(mammalName)

class NonWingedMammal(Mammal):
  def __init__(self, NonWingedMammal):
    print(NonWingedMammal, "can't fly.")
    super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
  def __init__(self, NonMarineMammal):
    print(NonMarineMammal, "can't swim.")
    super().__init__(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):
  def __init__(self):
    print('Dog has 4 legs.');
    super().__init__('Dog')

d = Dog()
print('')
bat = NonMarineMammal('Bat')
```

Output

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

Method Resolution Order (MRO)

Method Resolution Order (**MRO**) is the **order in which methods should be inherited in the presence of multiple inheritance**. You can view the MRO by using the ***__mro__*** **attribute**.

```
>>> Dog.__mro__
(<class 'Dog'>,
<class 'NonMarineMammal'>,
<class 'NonWingedMammal'>,
<class 'Mammal'>,
<class 'Animal'>,
<class 'object'>)
```

Here is how MRO works:

- A method in the derived calls is always called before the method of the base class. In our example, ***Dog*** **class** is called before ***NonMarineMammal*** or ***NoneWingedMammal***. These two classes are called before ***Mammal***, which is called before ***Animal***, and ***Animal*** **class** is called **before the object**.
- If there are multiple parents like ***Dog(NonMarineMammal, NonWingedMammal)***, methods of ***NonMarineMammal*** is **invoked first** because it **appears first**.

Python Operator Overloading

- You can change the meaning of an operator in Python depending upon the operands used.
- Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- This feature in Python that allows the same operator to have different meaning according to the context is called **operator overloading**.
- So, what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

- Here, we can see that a *TypeError* was raised, since Python didn't know how to add two Point objects together.
- However, we can achieve this task in Python through ***operator overloading***. But first, let's get a notion about special functions.

Python Special Functions

- Class functions that begin with double underscore __ are called special functions in Python.
- These functions are not the typical functions that we define for a class. The __init__() function we defined above is one of them. It gets called every time we create a new object of that class.
- There are numerous other special functions in Python. Visit Python Special Functions to learn more about them.
- Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the ***print()*** function to print the coordinates of the Point object instead of what we got. We can define a ***__str__()*** method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
class Point:
  def __init__(self, x = 0, y = 0):
    self.x = x
    self.y = y

  def __str__(self):
    return "({0},{1})".format(self.x,self.y)
```

Now let's try the ***print()*** function again.

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return "({0}, {1})".format(self.x, self.y)

p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function ***str()*** or ***format()***.

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you use ***str(p1)*** or ***format(p1)***, Python internally calls the ***p1.__str__()*** method. Hence the name, special functions.

Now let's go back to operator overloading.

Overloading the + Operator

To overload the + operator, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
  def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

Now let's try the addition operation again:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output

```
(3,5)
```

What actually happens is that, when you use ***p1 + p2***, Python calls ***p1.__add__(p2)*** which in turn is ***Point.__add__(p1,p2)***. After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	**Expression**	**Internally**
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)

Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators

- Python does not limit operator overloading to arithmetic operators only. We can **overload comparison operators as well.**
- Suppose we wanted to implement the less than symbol < symbol in our Point class.
- Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output
```
True
False
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	**Expression**	**Internally**
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)

Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Private Instance Variables (Encapsulation)

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g.***_spam***) should be treated as a **non-public part of the API** (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called ***name mangling***. Any identifier of the form ***__spam*** (**at least two leading underscores**, **at most one trailing underscore**) is textually replaced with ***_classname__spam***, where ***classname*** is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

Example

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        # The zip() function takes iterables (can be zero or more),
        # aggregates them in a tuple and return it.
        for item in zip(keys, values):
            self.items_list.append(item)

str='Gujarat'
x=Mapping(str)
print(x.items_list)
x.update('India')
print(x.items_list)

str='Surat'
y=MappingSubclass(str)
```

```
print(y.items_list)
y.update([1, 2, 3, 4], [1, 4, 9, 16])
print(y.items_list)
```

OUTPUT

```
['G', 'u', 'j', 'a', 'r', 'a', 't']
['G', 'u', 'j', 'a', 'r', 'a', 't', 'I', 'n', 'd', 'i', 'a']
['S', 'u', 'r', 'a', 't']
['S', 'u', 'r', 'a', 't', (1, 1), (2, 4), (3, 9), (4, 16)]
```

The above example would work even if ***MappingSubclass*** were to introduce a ***__update*** identifier since it is replaced with ***_Mapping__update*** in the ***Mapping*** class and ***_MappingSubclass__update*** in the ***MappingSubclass*** class respectively.

In the context of class, private means the attributes are only available for the members of the class not for the outside of the class.

Suppose we have the following class ***Student*** which has private attributes (***__school***):

```
class Student:
   def __init__(self, name, school):
      self.name=name          # public
      self.__school=school   # private

   def Print(self):
      print(f'Name: {self.name}')
      print(f'School: {self.__school}')
```

We create an instance of class ***Student***, then trying to access its attributes whether it's public or private:

```
std=Student(name='Rahul', school='PRK')
print(std.name)
print(std.school)
```

Output

```
Rahul
Traceback (most recent call last):
  File "D:\Personal\Ravi\Python\MCA\P021-6-2.py", line 16, in <module>
    print(std.school)
AttributeError: 'Student' object has no attribute 'school'
```

For public attribute ***name***, we can access through an instance variable, but not for the private attribute ***school***. Even we try this:

```
print(std.__school)
```

Output

```
Traceback (most recent call last):
  File "D:\Personal\Ravi\Python\MCA\P021-6-2.py", line 17, in <module>
    print(std.__school)
AttributeError: 'Student' object has no attribute '__school'
```

still we're not allowed to access it.

But One underscore ('_') **with the class name** will do magic:

```
print(std._Student__school)
```

Output

```
PRK
```

The following call also works as expected:

```
std.Print()
```

Output

```
Name: Rahul
School: PRK
```

Example

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print(f'Selling Price: {self.__maxprice}')

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()
```

```
# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a ***Computer*** class.

We used ***__init__()*** method to store the maximum selling price of ***Computer***. We tried to modify the price. However, we can't change it because Python treats the ***__maxprice*** as private attributes.

As shown, to change the value, we have to use a setter function i.e ***setMaxPrice()*** which takes price as a parameter.

Private Methods:

We're going to use the same code as in the previous section, but we will add two methods: ***Test***() and ***__Test()*** methods:

```
class Student:
    def __init__(self, name, school):
        self.name=name          # public
        self.__school=school    # private

    def Print(self):
        print(f'Name: {self.name}')
        print(f'School: {self.__school}')

    def __Test(self):
        print('This is private method')

    def Test(self):
        print('This is public method')
        self.__Test()
```

If we try to use the private method.

```
std=Student(name='Rahul', school='PRK')
print(std.name)
std.__Test()
```

Output

```
Rahul
Traceback (most recent call last):
  File "D:\Personal\Ravi\Python\MCA\P021-6-3.py", line 28, in <module>
    std.__Test()
AttributeError: 'Student' object has no attribute '__Test'
```

However, the following will work:

```
std=Student(name='Rahul', school='PRK')
print(std.name)
std.Test()
```

Output

```
Rahul
This is public method
This is private method
```

The right way of accessing private method is this:

```
std._Student__Test()
```

Output

```
This is private method
```

Of course, calling private method via public will work as expected:

```
Std.Test()
```

Output

```
This is public method
This is private method
```

Polymorphism

Polymorphism (having many forms) is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose we need to colour a shape, there are multiple shape options (rectangle, square, circle). However, we could use the same method to colour any shape. This concept is called **Polymorphism**.

Example of inbuilt Polymorphism:

```
print(len('VNSGU'))
print(len([1, 2, 3, 4]))
print(len({1, 2, 3, 4}))
```

Output

```
5
4
4
```

Examples of used defined polymorphic functions:

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
        return x + y + z

print(add(2, 3))
print(add(2, 3, 4))
```

Output

```
5
9
```

Polymorphism with class methods:

The code given below shows how python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Example

```
class Parrot:
    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Output

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes ***Parrot*** and ***Penguin***. Each of them have a common ***fly()*** method. However, their functions are different.

To use polymorphism, we created a common interface i.e. ***flying_test()*** function that takes any object and calls the object's ***fly()*** method. Thus, when we passed the ***blu*** and ***peggy*** objects in the ***flying_test()*** function, it ran effectively.

Polymorphism with Inheritance:

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

    class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects:

It is also possible to create a function that can take any object, allowing for polymorphism. In this example, let's create a function called "***func()***" which will take an object which we will name "***obj***". Though we are using the name '***obj***', any instantiated object will be able to be called into this function. Next, let's give the function something to do that uses the '***obj***' object we passed to it. In this case let's call the three methods, viz., ***capital()***, ***language()*** and ***type()***, each of which is defined in the two classes '***India***' and '***USA***'. Next, let's create instances of both the '***India***' and '***USA***' classes if we don't have them already. With those, we can call their action using the same ***func()*** function:

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

_Single Leading Underscores

One underscore in the beginning of a method, function, or data member means you shouldn't access this method because it's not part of the API. Let's look at this snippet of code:

Example

```
def _get_errors(self):
    if self._errors is None:
        self.full_clean()
    return self._errors

errors = property(_get_errors)
```

The snippet is taken from the Django source code (django/forms/forms.py). This suggests that ***errors*** is a property, and it's also a part of the API, but the method, ***_get_errors***, is "**private**", so one shouldn't access it.

__Double Leading Underscores

Two underlines, in the beginning, cause a lot of confusion. This is about syntax rather than a convention. double underscore will mangle the attribute names of a class to avoid conflicts of attribute names between classes. For example:

Example

```
class BaseClass:
    def _single_method(self):
        pass

    def __double_method(self): # for mangling
        pass

class SubClass(BaseClass):
    def __double_method(self): # for mangling
        pass
```

__Double leading and Double trailing underscores__

There's another case of double leading and trailing underscores. We follow this while using special variables or methods (called "magic method") such as__len__, __init__. These methods provide special syntactic features to the names. For example, __file__ indicates the location of the Python file, __eq__ is executed when a == b expression is executed.

Example

```
class Geek:

    # '__init__' for initializing, this is a
    # special method
    def __init__(self, ab):
        self.ab = ab

    # custom special method. try not to use it
    def __custom__(self):
        pass
```

Python Try Except

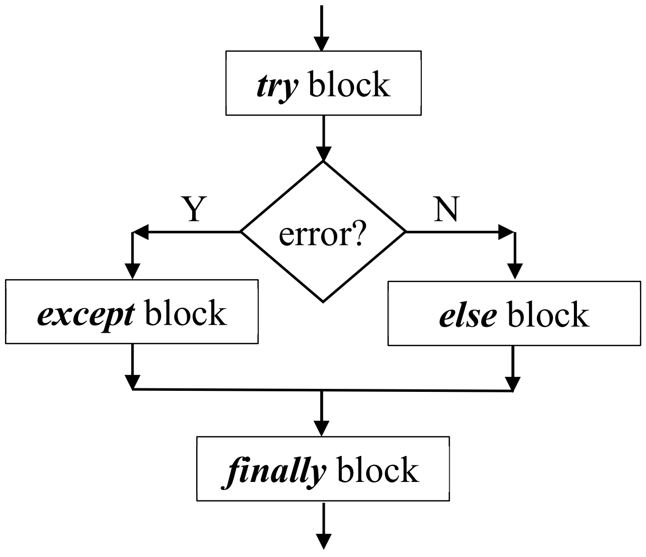
- The ***try*** block lets you test a block of code for errors.
- The ***except*** block lets you handle the error.
- The ***else*** block lets you define a block of code to be executed if no errors were raised
- The ***finally*** block lets you execute code, regardless of the result of the try- and except blocks.

try:

except:

else:

finally:



Exception Handling:

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

Example

The ***try*** block will generate an exception because x is not defined:

```
try:
        print(x)
except:
        print("An exception occurred")
```

Since the ***try*** block raises an error, the ***except*** block will be executed.

Without the ***try*** block, the program will crash and raise an error:

Example

This statement will raise an error because ***x*** is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the ***try*** block raises a ***NameError*** and another for other errors:

```
try:
        print(x)
except NameError:
        print("Variable x is not defined")
except:
        print("Something else went wrong")
```

Else

You can use the ***else*** keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the ***try*** block does not generate any error:

```
try:
        print("Hello")
except:
        print("Something went wrong")
else:
        print("Nothing went wrong")
```

Finally

The ***finally*** block, if specified, will be **executed regardless if the try block raises an error or not**.

Example

```
try:
        print(x)
except:
        print("Something went wrong")
finally:
        print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
        f = open("demofile.txt")
        f.write("Lorum Ipsum")
except:
        print("Something went wrong when writing to the file")
finally:
        f.close()
```

The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or ***raise***) an exception, use the ***raise*** keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
        raise Exception("Sorry, no numbers below zero")
```

The ***raise*** keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a ***TypeError*** if x is not an integer:

```
x = "hello"

if not type(x) is int:
        raise TypeError("Only integers are allowed")
```

Some of the common ***built-in exceptions in Python programming*** along with the error that cause them are listed below:

Exception	**Cause of Error**
AssertionError	Raised when an assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits end-of-file condition.

FloatingPointError	Raised when a floating-point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

If required, we can also define our own exceptions in Python.

Creating Custom Exceptions

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in ***Exception*** class. Most of the built-in exceptions are also derived from this class.

Example: User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, a hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions

class Error(Exception):
"""Base class for other exceptions"""
        pass

class ValueTooSmallError(Error):
"""Raised when the input value is too small"""
        pass

class ValueTooLargeError(Error):
"""Raised when the input value is too large"""
        pass

# you need to guess this number
number = 10

# user guesses a number until he/she gets it right
while True:
        try:
                i_num = int(input("Enter a number: "))
                if i_num < number:
                        raise ValueTooSmallError
                elif i_num > number:
                        raise ValueTooLargeError
                break
        except ValueTooSmallError:
                print("This value is too small, try again!")
                print()
        except ValueTooLargeError:
                print("This value is too large, try again!")
                print()

print("Congratulations! You guessed it correctly.")
```

Here is a sample run of this program.

```
Enter a number: 12
This value is too large, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

We have defined a base class called ***Error***.

The other two exceptions (***ValueTooSmallError*** and ***ValueTooLargeError***) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming, but you are not limited to this way only.

Customizing Exception Classes

We can further customize this class to accept other arguments as per our needs.

Let's look at one example:

```
class SalaryNotInRangeError(Exception):
"""Exception raised for errors in the input salary.

Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
"""

def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
        raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Here, we have overridden the constructor of the ***Exception*** class to accept our own custom arguments ***salary*** and ***message***. Then, the constructor of the parent ***Exception*** class is called manually with the ***self.message*** argument using ***super()***.

The custom ***self.salary*** attribute is defined to be used later.

The inherited ***__str__*** method of the ***Exception*** class is then used to display the corresponding message when ***SalaryNotInRangeError*** is raised.

We can also customize the ***__str__*** method itself by overriding it.

```
class SalaryNotInRangeError(Exception):
"""Exception raised for errors in the input salary.

        Attributes:
                salary -- input salary which caused the error
                message -- explanation of the error
"""

        def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
                self.salary = salary
                self.message = message
                super().__init__(self.message)

        def __str__(self):
                return f'{self.salary} -> {self.message}'


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
        raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "/home/bsoyuj/Desktop/Untitled-1.py", line 20, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: 2000 -> Salary is not in (5000, 15000) range
```

Python Iterators

Consider the following examples:

```
for element in [1, 2, 3]:
   print(element)
for element in (1, 2, 3):
   print(element)
for key in {'one':1, 'two':2}:
   print(key)
for char in "123":
   print(char)
for line in open("myfile.txt"):
   print(line, end='')
```

Python. Behind the scenes, the ***for*** statement calls ***iter()*** on the container object. The function returns an iterator object that defines the method ***__next__()*** which accesses elements in the container one at a time. When there are no more elements, ***__next__()*** raises a ***StopIteration*** exception which tells the ***for*** loop to terminate.

An **iterator** is an object that contains a countable number of values.

An iterator is 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__()***.

iter(object, sentinel)

Parameters:

1. ***object***: (**Required**) The object whose iterator should be created. If the second parameter ***sentinel*** is not specified, this object must be a collection object with the ***__iter__()*** method or the ***__getitem__()*** method with integer arguments starting at 0. If ***sentinel*** is given, then this must be a callable object. The ***__next__()*** function returns a value from the collection on each call.
2. ***sentinel***: (**Optional**) A value that indicates the end of sequence.

Return Value:

Returns an iterator object for the given object.

Iterators are implicitly used whenever we deal with collections of data types such as list, tuple or string (they are quite fittingly called ***iterables***). The usual method to traverse a collection is using the for loop, as shown below.

Example:

```
nums = [1, 2, 3, 4, 5]
for item in nums:
```

```
            print(item)

    Output
    1
    2
    3
    4
    5
```

In the above example, the for loop iterates over a list object- *nums* and prints each individual element. When we use a for loop to traverse any **iterable** object, internally it uses the ***iter()*** method, same as below.

```
def traverse(iterable):
   it=iter(iterable)
   while True:
      try:
         item=next(it)
         print (item)
      except StopIteration:
         break
```

Iterator vs Iterable

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:

Example:

Return an iterator from a tuple, and print each value:

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

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

Even strings are iterable objects, and can return an iterator:

Example:

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)
```

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

Looping Through an Iterator

We can also use a ***for*** loop to iterate through an iterable object:

Example:

Iterate the values of a tuple:

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

for x in mytuple:
        print(x)
```

Example:

Iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
        print(x)
```

The *for* loop actually creates an iterator object and executes the *next()* method for each loop.

Create an Iterator

- To create an object/class as an iterator you have to implement the methods ***__iter__()*** and ***__next__()*** to your object.
- As you have learned in the Python Classes/Objects chapter, all classes have a function called ***__init__()***, which allows you to do some initializing when the object is being created.
- The ***__iter__()*** method acts similar, 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.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1, 2, 3, 4, 5 etc.):

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough ***next()*** statements, or if it was used in a ***for*** loop.

To prevent the iteration to go on forever, we can use the ***StopIteration*** statement.

In the ***__next__()*** method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
```

```
                self.a += 1
                return x
            else:
                raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

Using Sentinel Parameter

The ***sentinel*** parameter is used to indicate the end of the sequence. However, the class must be callable, which internally calls __next__() method. The following DataStore class is modified to demonstrate the use of the sentinel parameter by adding __call__ = __next__.

Example:

```
class DataStore:
    def __init__(self, data):
        self.index = -1
        self.data = data
    def __iter__(self):
        return self
    def __next__(self):
        if (self.data[self.index] == self.data or self.index==len(self.data)-1):
            raise StopIteration
        self.index +=1
        return self.data[self.index]
    __call__ = __next__

ds = DataStore([1,2,3])
itr = iter(ds, 3) # sentinel is 3, so it will stop when encounter 3

for i in itr:
    print(i)

Output
1
2
```

NumPY

- NumPy is a Python package. It stands for **'Numerical Python'**. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.
- Using NumPy, a developer can perform the following operations:
    - Mathematical and logical operations on arrays.
    - Fourier transforms and routines for shape manipulation.
    - Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.
- NumPy is often used along with packages like **SciPy** (**Scientific Python**) and **Mat−plotlib** (plotting library). This combination is widely used as a replacement for **MatLab**, a popular platform for technical computing.
- It is open source, which is an added advantage of NumPy.

Installation:

```
pip install numpy
```

here ***pip*** is popular Python Package Installer

To test whether NumPy module is properly installed, try to import it from Python prompt.

```
import numpy
```

If it is not installed, the following error message will be displayed.

```
Traceback (most recent call last):
        File "<pyshell#0>", line 1, in <module>
                import numpy
ImportError: No module named 'numpy'
```

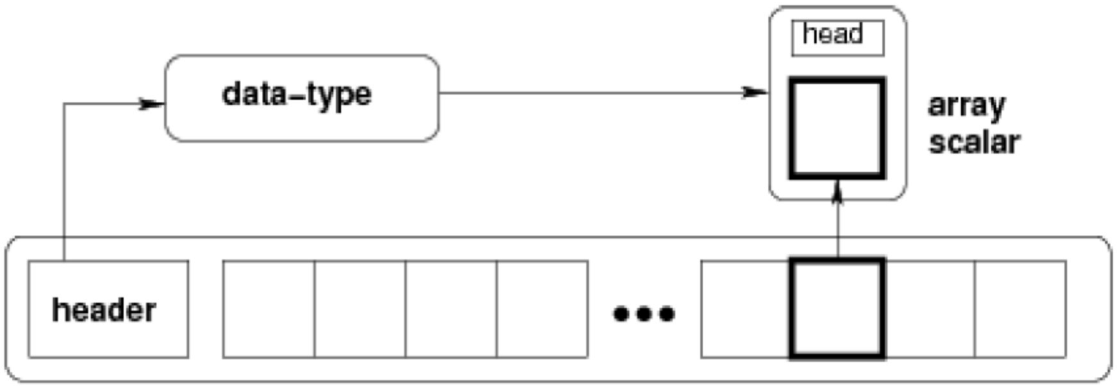
Alternatively, NumPy package is imported using the following syntax:

```
import numpy as np
```

NumPy Array

ndarray – N-dimensional array type

- describes collection of items of same type
- items can be accessed using a zero-based index
- every item takes same size of block in the memory
- each element is an object of data type object called ***dtype***
- Any item extracted from ***ndarray*** object (by slicing) is represented by a Python object of one of ***array scalar types***.
- The following diagram shows a relationship between ***ndarray***, data type object (***dtype***) and ***array scalar type***:



ndarray

An instance of ***ndarray*** class can be constructed by different array creation routines described later in the tutorial. The basic ***ndarray*** is created using an array function in NumPy as follows:

numpy.array

Following are the important attributes of a ***ndarray*** object:

ndim	the number of axes (dimensions) of the array.
	`>>> x=np.array([1, 2, 3])`
	`>>> x.ndim`
	`1`
	`>>> x=np.array([[1, 2],`
	`[3, 4],`
	`[5, 6]])`
	`>>> x.ndim`
	`2`

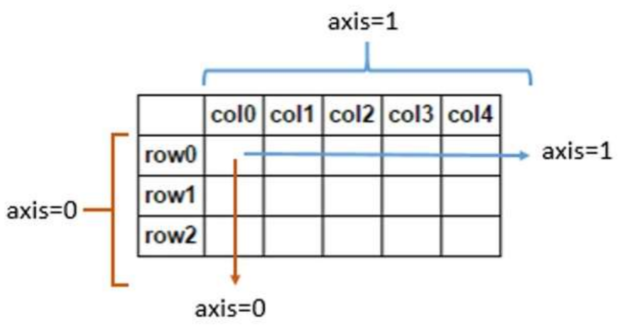
shape	the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a **matrix with *n* rows and *m* columns**, shape will be **(*n*, *m*)**. The length of the shape tuple is therefore the number of axes, *ndim*.

	>>> x=np.array([1, 2, 3]) *>>> x.shape* *(3,)* *>>> x=np.array([[1, 2],* *[3, 4],* *[5, 6]])* *>>> x.shape* *(3, 2)*
size	the **total number of elements** of the array. This is equal to the **product of the elements of shape**. *>>> x=np.array([1, 2, 3])* *>>> x.size* *3* *>>> x=np.array([[1, 2],* *[3, 4],* *[5, 6]])* *>>> x.size* *6*

dtype	an object describing the type of the elements in the array. One can create or specify *dtype*'s using standard Python types. Additionally, NumPy provides types of its own. *numpy.int32*, *numpy.int16*, and *numpy.float64* are some examples.
	>>> x=np.array([1, 2, 3], dtype=np.float64)
	>>> x
	array([1., 2., 3.])
	
	>>> dt = np.dtype('i4') *# 32-bit signed integer*
	>>> dt = np.dtype('f8') *# 64-bit floating-point number*
	>>> dt = np.dtype('c16') *# 128-bit complex floating-point number*
	>>> dt = np.dtype('a25') *# 25-length zero-terminated bytes*
	>>> dt = np.dtype('U25') *# 25-character string*

itemsize	the **size in bytes of each element** of the array. For example, an array of elements of type *float64* has itemsize *8 (=64/8)*, while one of type *complex32* has itemsize *4 (=32/8)*. It is equivalent to *ndarray.dtype.itemsize*. *>>> x=np.array([1., 2., 3.])* *>>> x.itemsize* *8* *>>> x.dtype.itemsize* *8*
data	the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities. *>>> x=np.array([1, 2, 3])* *>>> x.data* *<memory at 0x0000026B49B551C0>* *>>> x=np.array([[1, 2],*

	[3, 4], *[5, 6]])* *>>> x.data* *<memory at 0x0000026B49A871E0>*



numpy.array()

Transforms sequences into one-dimensional array, sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three dimensional arrays, and so on.

numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)

Parameters

object: array_like
An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype: data-type, optional
The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

copy: bool, optional
If **true** (**default**), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`dtype`, `order`, etc.).

order : {'K', 'A', 'C', 'F'}, optional
Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major).

If object is an array the following holds.

order	no copy	Copy=True
'K'	unchanged	F & C order preserved, otherwise most similar order
'A'	unchanged	F order if input is F and not C, otherwise C order
'C'	C order	C order

‘F’	F order	F order

When ``copy=False`` and a copy is made for other reasons, the result is the same as if ``copy=True``, with some exceptions for `A`, see the Notes section. The default order is 'K'.

subok : bool, optional
If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

ndmin : int, optional
Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

like : array_like
Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as ``like`` supports the ``__array_function__`` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

Functions:

array	Transforms sequences into one-dimensional array, sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three dimensional arrays, and so on. ***numpy. array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)*** The type of the array can also be explicitly specified at creation time: *c = np.array([[1,2], [3,4]], dtype=complex)*
arange	creates sequences of numbers, which is analogous to the Python built-in ***range***, but returns an **array**. We can specify the **first number**, **last number**, and the **step size.** ***numpy.arange([start,] stop [, step,], dtype=None)*** *>>> np.arange(2, 9, 2)* *array([2, 4, 6, 8])* Note: When ***arange*** is used with floating-point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating-point precision. For this reason, it is usually better to use the function ***linspace*** that receives as an argument the number of elements that we want, instead of the step. e.g. *>>>np.linspace(0, 2, 9) # 9 numbers from 0 to 2* *array([0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2.])*

linspace	creates an array with values that are spaced linearly in a specified interval:
	numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
	
	>>> np.linspace(0, 10, num=5)
	array([0. , 2.5, 5. , 7.5, 10.])

reshape	Returns an array containing the same data with a new shape. ***ndarray.reshape(shape, order='C')*** ***ndarray.reshape(-1)***: it will flatten the array, i.e. any multidimensional array will be reshaped to 1 D array. *>>> x=np.arange(6)* *>>> x* *array([0, 1, 2, 3, 4, 5])* *>>> x.reshape((2, 3))* *array([[0, 1, 2],* *[3, 4, 5]])* *>>> x.reshape((3, 2))* *array([[0, 1],* *[2, 3],* *[4, 5]])* *>>> x.reshape((2, 3), order='C') # Row-Major* *array([[0, 1, 2],* *[3, 4, 5]])* *>>> x.reshape((2, 3), order='F') # Column-Major* *array([[0, 2, 4],* *[1, 3, 5]])+* Order: 'C' means C like index (Row-Major) 'F' means ForTran like index (Column-Major)

resize	Change shape and size of array in-place. ***ndarray.resize(new_shape, refcheck=True)*** ***new_shape***: *tuple of ints, or n ints* (Shape of resized array.) ***refcheck***: *bool, optional* (If False, reference count will not be checked. Default is True.) Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped: *>>> a = np.array([[0, 1], [2, 3]], order='C')* *>>> a.resize((2, 1))* *>>> a* *array([[0],* *[1]])* *>>> a = np.array([[0, 1], [2, 3]], order='F')* *>>> a.resize((2, 1))* *>>> a* *array([[0],* *[2]])* Enlarging an array: as above, but missing entries are filled with zeros: *>>> b = np.array([[0, 1], [2, 3]])* *>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple* *>>> b* *array([[0, 1, 2],* *[3, 0, 0]])* Referencing an array prevents resizing. *>>> c = a* *>>> a.resize((1, 1))* *Traceback (most recent call last):* *...* *ValueError: cannot resize an array that references or is referenced* *...* Unless ***refcheck*** is False: *>>> a.resize((1, 1), refcheck=False)* *>>> a* *array([[0]])* *>>> c* *array([[0]])*

concatenate	Join arrays by putting two or more arrays in a single array. It will join by axes (default axes=0) *numpy.concatenate((a1, a2, ...), axis=0, out=None, dtype=None, casting="same_kind")* *>>> a = np.array([[1, 2], [3, 4]])* *>>> b = np.array([[5, 6]])* *>>> np.concatenate((a, b), axis=0)* *array([[1, 2],* *[3, 4],* *[5, 6]])* *>>> np.concatenate((a, b.T), axis=1)* *array([[1, 2, 5],* *[3, 4, 6]])* *>>> np.concatenate((a, b), axis=None)* *array([1, 2, 3, 4, 5, 6])*
zeros	creates an array full of zeros. By default, the *dtype* of the created array is *float64*. ***numpy.zeros(shape, dtype=float, order='C')*** *order : {'C', 'F'}, optional, default: 'C'* *Whether to store multi-dimensional data in **row-major (C-style)** or **column-major (Fortran-style)** order in memory.* *>>> np.zeros(2)* *array([0., 0.])*
ones	creates an array full of ones. By default, the *dtype* of the created array is *float64*. ***numpy.ones(shape, dtype=float, order='C')*** *order : {'C', 'F'}, optional, default: 'C'* *Whether to store multi-dimensional data in **row-major (C-style)** or **column-major (Fortran-style)** order in memory.* *>>> np.ones(2)* *array([1., 1.])*
empty	creates an array whose initial content is random and depends on the state of the memory. By default, the *dtype* of the created array is *float64*. ***numpy.empty(shape, dtype=float, order='C')*** *order : {'C', 'F'}, optional, default: 'C'* *Whether to store multi-dimensional data in **row-major (C-style)** or **column-major (Fortran-style)** order in memory.* *>>> # Create an empty array with 2 elements* *>>> np.empty(2)* *array([3.14, 42.]) # may vary*

fromiter	This function builds an ***ndarray*** object from any ***iterable*** object. A new one-dimensional array is returned by this function. ***numpy.fromiter(iterable, dtype, count=-1)*** where: ***iterable*** any iterable object ***dtype*** data type of resultant array (compulsory) ***count*** number of items to be read from iterator. Default is -1 which means all data to be read *>>>lst=[1, 2, 3, 4, 5] # list* *>>>x=np.fromiter(lst, dtype=int)* *>>>print(x)* *[1 2 3 4 5]* *>>>print(x.size)* *5* *>>>x= np.fromiter(lst, dtype=int32, count=3)* *>>>print(x)* *[1 2 3]* *>>>print(x.size)* *3* *>>>str='12345' # string* *>>>x=np.fromiter(str, dtype=int)* *>>>print(x)* *[1 2 3 4 5]*

sort	Return a sorted copy of an array. ***np.sort(a, axis=-1, kind=None, order=None)*** ***a***: array to be sorted ***axis***: ***int*** or ***None***, optional Axis along which to sort. If ***None***, the **array is flattened before sorting**. The **default is -1**, which **sorts along the last axis**. ***kind***: ***{'quicksort', 'mergesort', 'heapsort', 'stable'},*** optional Sorting algorithm. The **default** is '***quicksort***'. Note that both '***stable***' and '***mergesort***' use **timsort** or **radix** sort under the covers and, in general, the actual implementation will vary with data type. The '***mergesort***' option is retained for backwards compatibility. ***order***: ***str*** or ***list of str,*** optional When ***a*** is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the ***dtype***, to break ties. *>>>a = np.array([[1,4],[3,1]])* *>>>np.sort(a) # sort along the last axis* *array([[1, 4],* * [1, 3]])* *>>> np.sort(a, axis=**None**) # sort the flattened array* *array([1, 1, 3, 4])* *>>> np.sort(a, axis=0) # sort along the first axis* *array([[1, 1],* * [3, 4]])* *>>> x=np.array([[9, 1, 8],* * [2, 4, 3],* * [7, 5, 6]])* *>>> np.sort(x)* *array([[1, 8, 9],* * [2, 3, 4],* * [5, 6, 7]])* *>>> np.sort(x, axis=None)* *array([1, 2, 3, 4, 5, 6, 7, 8, 9])* *>>> np.sort(x, axis=0)* *array([[2, 1, 3],* * [7, 4, 6],* * [9, 5, 8]])*

Few Examples:

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
```

```
array([2, 3, 4])
>>> a.dtype
dtype('int32')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')


>>> a = np.array(1,2,3,4) # WRONG
Traceback (most recent call last):
...
TypeError: array() takes from 1 to 2 positional arguments but 4 were given
>>> a = np.array([1,2,3,4]) # RIGHT


>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[1.5, 2. , 3. ],
[4. , 5. , 6. ]])


>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
        [3.+0.j, 4.+0.j]])


>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])

>>> np.ones( (2,3,4), dtype=np.int16 ) # dtype can also be specified
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],
        [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)

>>> np.empty( (2,3) )                              # uninitialized
array([[ 3.73603959e-262, 6.02658058e-154, 6.55490914e-260], # may vary
        [ 5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])

>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])

>>> np.arange( 0, 2, 0.3 ) # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 ) # 9 numbers from 0 to 2
array([0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
>>> x = np.linspace( 0, 2*pi, 100 ) # useful to evaluate function at lots of points
>>> f = np.sin(x)
```

numpy.asarray() in Python

numpy.asarray() function is used when we want to convert input to an array. Input can be lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

Syntax : ***numpy.asarray(arr, dtype=None, order=None)***

Parameters :

arr :
[array_like] Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

dtype :
[data-type, optional] By default, the data-type is inferred from the input data.

order :
Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

Return :
[ndarray] Array interpretation of arr. No copy is performed if the input is already ndarray with matching dtype and order. If arr is a subclass of ndarray, a base class ndarray is returned.

Example

```
tpl=(1, 2, 3, 4, 5)
a=np.asarray(tpl)
print(a)
```

Output

```
[1 2 3 4 5]
```

Example

```
lst=[1, 2, 3, 4, 5]
a=np.asarray(lst)
print(a)
```

Output

```
[1 2 3 4 5]
```

Printing Arrays

One-dimensional arrays are then printed as rows, bi-dimensional as matrices and tri-dimensional as lists of matrices.

```
>>> a = np.arange(6) # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>

>>> b = np.arange(12).reshape(4,3) # 2d array
>>> print(b)
[[ 0 1 2]
[ 3 4 5]
[ 6 7 8]
[ 9 10 11]]
>>>

>>> c = np.arange(24).reshape(2,3,4) # 3d array
>>> print(c)
[[[ 0 1 2 3]
[ 4 5 6 7]
[ 8 9 10 11]]

[[12 13 14 15]
[16 17 18 19]
[20 21 22 23]]]
```

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> print(np.arange(10000))
[ 0 1 2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[ 0 1 2 ... 97 98 99]
[ 100 101 102 ... 197 198 199]
[ 200 201 202 ... 297 298 299]
...
[9700 9701 9702 ... 9797 9798 9799]
[9800 9801 9802 ... 9897 9898 9899]
```

```
                    [9900 9901 9902 ... 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using ***set_printoptions***.

```
>>> np.set_printoptions(threshold=sys.maxsize) # sys module should be imported
```

Basic Operations

- Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False])
```

- Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
>>> A = np.array( [[1,1],
                   [0,1]] )
>>> B = np.array( [[2,0],
                   [3,4]] )
>>> A * B # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B) # another matrix product
array([[5, 4],
       [3, 4]])
```

- Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

- When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behaviour known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([1. , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
-0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

- Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ***ndarray*** class.

```
>>> a = rg.random((2,3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
        [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]])
>>>
>>> b.sum(axis=0) # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1) # min of each row
array([0, 4, 8])
>>>
```

```
>>> b.cumsum(axis=1) # cumulative sum along each row
array([[ 0, 1, 3, 6],
       [ 4, 9, 15, 22],
       [ 8, 17, 27, 38]])
```

Converting Data Type of Elements of Existing Arrays

Make a copy of the array with the ***astype()*** method to change the data type of an existing array.

The ***astype()*** function creates a copy of the array and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like ***float*** for float and ***int*** for integer.

Example

Change data type from float to integer by using ***'i'*** as parameter value:

```
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

Convert a 1D array into a 2D array (how to add a new axis to an array)

You can use ***np.newaxis*** and ***np.expand_dims*** to **increase** the dimensions of your existing array.

Using ***np.newaxis*** will **increase** the dimensions of your array **by one dimension** when used once. This means that a **1D** array will become a **2D** array, a **2D** array will become a **3D** array, and so on.

For example, if you start with this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
```

You can use ***np.newaxis*** to add a new axis:

```
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
```

You can explicitly convert a 1D array with either a **row vector or a column vector** using ***np.newaxis***.

For example, you can convert a 1D array to a **row vector** by **inserting** an axis **along the first dimension**:

```
>>> row_vector = a[np.newaxis, :]
>>> row_vector.shape
(1, 6)
```

Or, for a **column vector**, you can **insert** an axis **along the second dimension**:

```
>>> col_vector = a[:, np.newaxis]
>>> col_vector.shape
(6, 1)
```

You can also **expand** an array by **inserting a new axis at a specified position** with ***np.expand_dims***.

For example, if you start with this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
```

You can use ***np.expand_dims*** to **add** an axis **at index position 1** with:

```
>>> b = np.expand_dims(a, axis=1)
>>> b.shape
(6, 1)
```

You can **add** an axis **at index position 0** with:

```
>>> c = np.expand_dims(a, axis=0)
>>> c.shape
(1, 6)
```

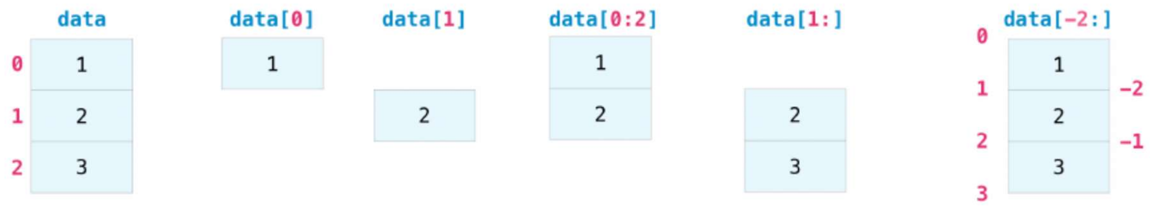
Indexing and slicing

You can **index** and **slice** NumPy arrays in the **same ways** you can **slice Python lists**.

```
>>> data = np.array([1, 2, 3])
>>> data[1]
```

```
2
>>> data[0:2]
array([1, 2])
>>> data[1:]
array([2, 3])
>>> data[-2:]
array([2, 3])
```

You can visualize it this way:



You may want to take a section of your array or specific array elements to use in further analysis or additional operations.

To do that, you'll need to subset, slice, and/or index your arrays.
If you want to select values from your array that fulfil certain conditions, it's straightforward with NumPy.

For example, if you start with this array:

```
>>> a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can easily print all of the values in the array that are less than 5.

```
>>> print(a[a < 5])
[1 2 3 4]
```

You can also select, for example, numbers that are equal to or greater than 5, and use that condition to index an array.

```
>>> five_up = (a >= 5)
>>> print(a[five_up])
[ 5 6 7 8 9 10 11 12]
```

You can select elements that are divisible by 2:

```
>>> divisible_by_2 = a[a%2==0]
>>> print(divisible_by_2)
[ 2 4 6 8 10 12]
```

Or you can select elements that satisfy two conditions using the & and | operators:

```
>>> c = a[(a > 2) & (a < 11)]
>>> print(c)
[ 3 4 5 6 7 8 9 10]
```

You can also make use of the logical operators **&** and **|** in order to return Boolean values that specify whether or not the values in an array fulfil a certain condition. This can be useful with arrays that contain names or other categorical values.

```
>>> five_up = (a > 5) | (a == 5)
>>> print(five_up)
[[False False False False]
[ True True True True]
[ True True True True]]
```

You can also use np.nonzero() to select elements or indices from an array.

Starting with this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use np.nonzero() to print the indices of elements that are, for example, less than 5:

```
>>> b = np.nonzero(a < 5)
>>> print(b)
(array([0, 0, 0, 0]), array([0, 1, 2, 3]))
```

In this example, a tuple of arrays was returned: one for each dimension. The first array represents the row indices where these values are found, and the second array represents the column indices where the values are found.

If you want to generate a list of coordinates where the elements exist, you can zip the arrays, iterate over the list of coordinates, and print them. For example:

```
>>> list_of_coordinates= list(zip(b[0], b[1]))
>>> for coord in list_of_coordinates:
... print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

You can also use np.nonzero() to print the elements in an array that are less than 5 with:

```
>>> print(a[b])
[1 2 3 4]
```

If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty. For example:

```
>>> not_there = np.nonzero(a == 42)
>>> print(not_there)
(array([], dtype=int64), array([], dtype=int64))


>>> a = np.arange(10)**3
>>> a
array([ 0, 1, 8, 27, 64, 125, 216, 343, 512, 729])

>>> a[2]
8

>>> a[2:5]
array([ 8, 27, 64])

# equivalent to a[0:6:2] = 1000;
# from start to position 6, exclusive, set every 2nd element to 1000
>>> a[:6:2] = 1000
>>> a
array([1000, 1, 1000, 27, 1000, 125, 216, 343, 512, 729])

>>> a[ : :-1] # reversed a
array([ 729, 512, 343, 216, 125, 1000, 27, 1000, 1, 1000])


>>> def f(x,y):
        ... return 10*x+y

>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0, 1, 2, 3],
        [10, 11, 12, 13],
        [20, 21, 22, 23],
        [30, 31, 32, 33],
        [40, 41, 42, 43]])

>>> b[2,3]
23

>>> b[0:5, 1] # each row in the second column of b
array([ 1, 11, 21, 31, 41])

>>> b[ : ,1] # equivalent to the previous example
array([ 1, 11, 21, 31, 41])

>>> b[1:3, : ] # each column in the second and third row of b
array([[10, 11, 12, 13],
        [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
>>> b[-1]        # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

The expression within brackets in b[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes. NumPy also allows you to write this using dots as b[i,...].

The **dots** (...) represent as many colons as needed to produce a complete indexing tuple. For example, if x is an array with 5 axes, then

- x[1,2,...] is equivalent to x[1,2,:,:,:],
- x[...,3] to x[:,:,:,:,3] and
- x[4,...,5,:] to x[4,:,:,5,:].

```
>>> c = np.array( [[[ 0, 1, 2], # a 3D array (two stacked 2D arrays)
                  ... [ 10, 12, 13]],
                  ... [[100,101,102],
                  ... [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...] # same as c[1,:,:] or c[1]
array([[100, 101, 102],
        [110, 112, 113]])
>>> c[...,2] # same as c[:,:,2]
array([[ 2, 13],
        [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
        ... print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
        ... print(element)
```

...
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43

Create an array from existing data (Slicing)

You can easily use create a new array from a section of an existing array.

Let’s say you have this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

You can create a new array from a section of your array any time by specifying where you want to slice your array.

```
>>> arr1 = a[3:8]
>>> arr1
array([4, 5, 6, 7, 8])
```

Here, you grabbed a section of your array from index position 3 through index position 8.

You can also ***stack*** two existing arrays, both vertically and horizontally. Let’s say you have two arrays, a1 and a2:

```
>>> a1 = np.array([[1, 1],
                   [2, 2]])
>>> a2 = np.array([[3, 3],
                   [4, 4]])
```

You can stack them vertically with ***vstack***:

```
>>> np.vstack((a1, a2))
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
```

Or stack them horizontally with ***hstack***:

```
>>> np.hstack((a1, a2))
array([[1, 1, 3, 3],
       [2, 2, 4, 4]])
```

You can **split** an array into several smaller arrays using hsplit. You can specify either the number of equally shaped arrays to return or the columns *after* which the division should occur.

Let’s say you have this array:

```
>>> x = np.arange(1, 25).reshape(2, 12)
>>> x
array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
        [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

Splitting

Split an array into multiple sub-arrays of equal size.

numpy.array_split(ary, indices_or_sections, axis=0)

```
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
[array([0.,  1.,  2.]), array([3.,  4.,  5.]), array([6.,  7.])]

>>> x = np.arange(9)
>>> np.array_split(x, 4)
[array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]
```

If you wanted to **split** this array into three equally shaped arrays, you would run:

```
>>> np.hsplit(x, 3)
[array([[1, 2, 3, 4],
        [13, 14, 15, 16]]), array([[ 5, 6, 7, 8],
        [17, 18, 19, 20]]), array([[ 9, 10, 11, 12],
        [21, 22, 23, 24]])]
```

If you wanted to **split** your array after the third and fourth column, you'd run:

```
>>> np.hsplit(x, (3, 4))
[array([[1, 2, 3],
        [13, 14, 15]]), array([[ 4],
        [16]]), array([[ 5, 6, 7, 8, 9, 10, 11, 12],
        [17, 18, 19, 20, 21, 22, 23, 24]])]
```

You can use the **view** method to create a new array object that looks at the same data as the original array (a ***shallow copy***).

Views are an important NumPy concept! NumPy functions, as well as operations like indexing and slicing, will return views whenever possible. This saves memory and is faster (no copy of the data has to be made). However, it's important to be aware of this - modifying data in a view also modifies the original array!

Let's say you create this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Now we create an array b1 by slicing a and modify the first element of b1. This will modify the corresponding element in a as well!

```
>>> b1 = a[0, :]
>>> b1
array([1, 2, 3, 4])
>>> b1[0] = 99
>>> b1
array([99, 2, 3, 4])
>>> a
array([[99, 2, 3, 4],
        [ 5, 6, 7, 8],
        [ 9, 10, 11, 12]])
```

Copies and Views

- The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
- The copy owns the data and any changes made to the copy will not affect the original array, and any changes made to the original array will not affect the copy.
- The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

View or Shallow Copy

Different array objects can share the same data. The view method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c = c.reshape((2, 6)) # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0, 4] = 1234 # a's data changes
>>> a
array([[ 0, 1, 2, 3],
        [1234, 5, 6, 7],
        [ 8, 9, 10, 11]])
```

Slicing an array returns a view of it:

```
>>> s = a[ : , 1:3]
>>> s[:] = 10 # s[:] is a view of s. Note the difference between s = 10 and s[:] = 10
>>> a
array([[ 0, 10, 10, 3],
        [1234, 10, 10, 7],
        [ 8, 10, 10, 11]])
```

Deep Copy

Using the copy method will make a complete copy of the array and its data (a *deep copy*). To use this on your array, you could run:

```
>>> d = a.copy() # a new array object with new data is created
>>> d is a
False
>>> d.base is a # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[ 0, 10, 10, 3],
        [1234, 10, 10, 7],
        [ 8, 10, 10, 11]])
```

Sometimes copy should be called after slicing if the original array is not required anymore. For example, suppose a is a huge intermediate result and the final result b only contains a small fraction of a, a deep copy should be made when constructing b with slicing:

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a # the memory of ``a`` can be released.
```

If b = a[:100] is used instead, a is referenced by b and will persist in memory even if del a is executed.

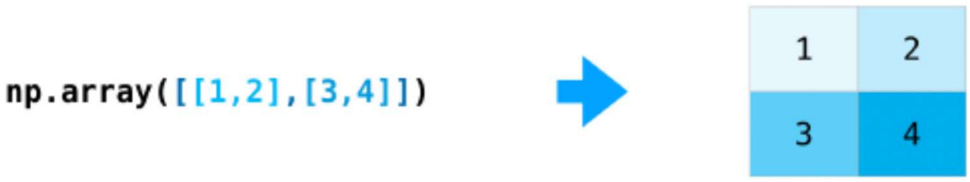
Statement	***b is a***	***b.flags.owndata***	***b.base is a***
b = a	*True*	*True*	*False*
b = a.view()	*False*	*False*	*True*
b = a[x, y : r, s] (slicing)	*False*	*False*	*True*
b = a.copy()	*False*	*True*	*False*

More useful array operations

```
>>> a = a=np.array([[20, 15, 25, 10],
                     [30, 45, 35, 40],
                     [5, 50, 65, 60]])
```

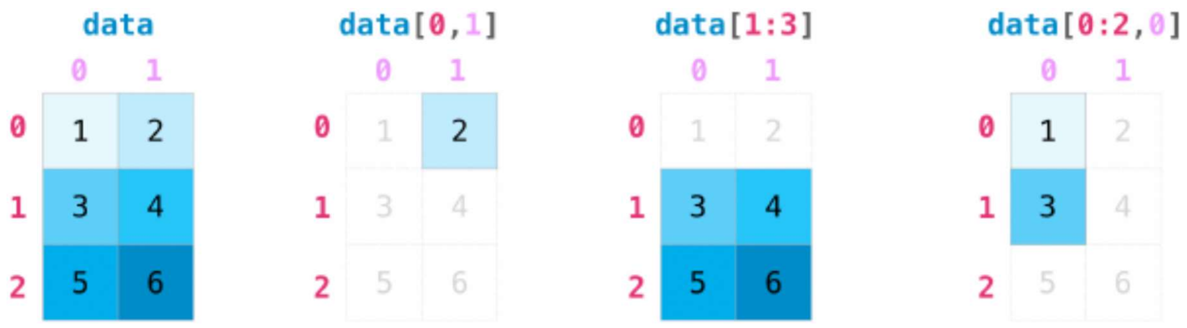
max()	a.max() = 65 a.max(axis=0)= [30 50 65 60] a.max(axis=1)= [25 45 65]
min()	a.min() = 5 a.min(axis=0)= [5 15 25 10] a.min(axis=1)= [10 30 5]
sum()	a.sum() = 400 a.sum(axis=0)= [55 110 125 110] a.sum(axis=1)= [70 150 180]

```
>>> data = np.array([[1, 2], [3, 4]])
>>> data
array([[1, 2],
       [3, 4]])
```



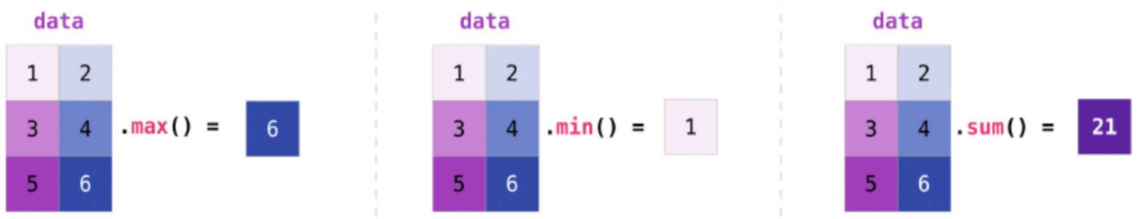
Indexing and slicing operations are useful when you're manipulating matrices:

```
>>> data[0, 1]
2
>>> data[1:3]
array([[3, 4]])
>>> data[0:2, 0]
array([1, 3])
```



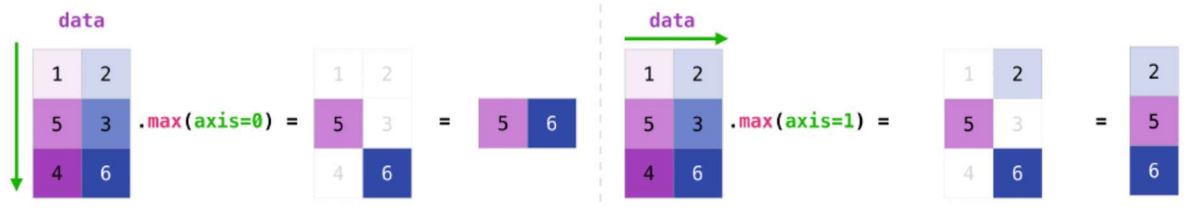
You can aggregate matrices the same way you aggregated vectors:

```
>>> data.max()
4
>>> data.min()
1
>>> data.sum()
10
```



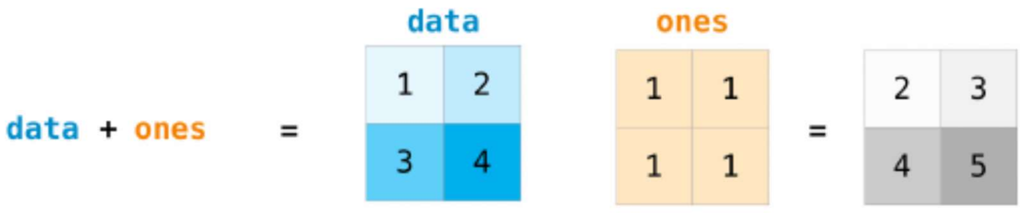
You can aggregate all the values in a matrix, and you can aggregate them across columns or rows using the axis parameter:

```
>>> data.max(axis=0)
array([3, 4])
>>> data.max(axis=1)
array([2, 4])
```



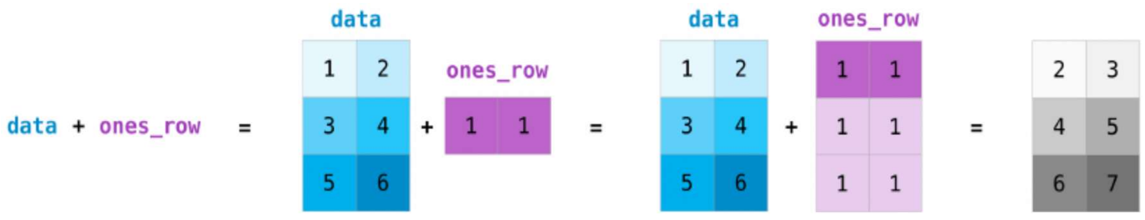
Once you've created your matrices, you can add and multiply them using arithmetic operators if you have two matrices that are the same size.

```
>>> data = np.array([[1, 2], [3, 4]])
>>> ones = np.array([[1, 1], [1, 1]])
>>> data + ones
array([[2, 3],
          [4, 5]])
```



You can do these arithmetic operations on matrices of different sizes, but only if one matrix has only one column or one row. In this case, NumPy will use its broadcast rules for the operation.

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> ones_row = np.array([[1, 1]])
>>> data + ones_row
array([[2, 3],
       [4, 5],
       [6, 7]])
```



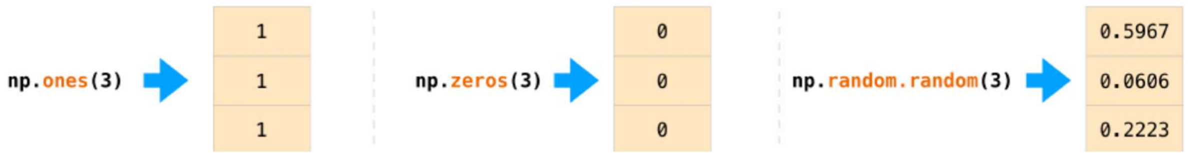
$$\texttt{data + ones\_row} = \begin{bmatrix}1&2\\3&4\\5&6\end{bmatrix} + \begin{bmatrix}1&1\end{bmatrix} = \begin{bmatrix}1&2\\3&4\\5&6\end{bmatrix} + \begin{bmatrix}1&1\\1&1\\1&1\end{bmatrix} = \begin{bmatrix}2&3\\4&5\\6&7\end{bmatrix}$$

Be aware that when NumPy prints N-dimensional arrays, the last axis is looped over the fastest while the first axis is the slowest. For instance:

```
>>> np.ones((4, 3, 2))
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],
        [[1., 1.],
        [1., 1.],
        [1., 1.]],
        [[1., 1.],
        [1., 1.],
        [1., 1.]],
        [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

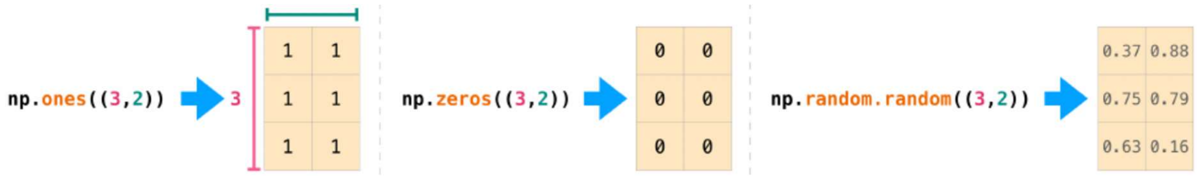
There are often instances where we want NumPy to initialize the values of an array. NumPy offers functions like ones() and zeros(), and the random.Generator class for random number generation for that. All you need to do is pass in the number of elements you want it to generate:

```
>>> np.ones(3)
array([1., 1., 1.])
>>> np.zeros(3)
array([0., 0., 0.])
# the simplest way to generate random numbers
>>> rng = np.random.default_rng(0)
>>> rng.random(3)
array([0.63696169, 0.26978671, 0.04097352])
```



You can also use ones(), zeros(), and random() to create a 2D array if you give them a tuple describing the dimensions of the matrix:

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> rng.random((3, 2))
array([[0.01652764, 0.81327024],
       [0.91275558, 0.60663578],
       [0.72949656, 0.54362499]]) # may vary
```



Get unique items and counts

You can find the unique elements in an array easily with ***np.unique***.

For example, if you start with this array:

```
>>> a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
```

you can use ***np.unique*** to print the unique values in your array:

```
>>> unique_values = np.unique(a)
>>> print(unique_values)
[11 12 13 14 15 16 17 18 19 20]
```

To get the indices of unique values in a NumPy array (an array of first index positions of unique values in the array), just pass the return_index argument in np.unique() as well as your array.

```
>>> unique_values, indices_list = np.unique(a, return_index=True)
>>> print(indices_list)
[ 0 2 3 4 5 6 7 12 13 14]
```

You can pass the return_counts argument in np.unique() along with your array to get the frequency count of unique values in a NumPy array.

```
>>> unique_values, occurrence_count = np.unique(a, return_counts=True)
>>> print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

This also works with 2D arrays! If you start with this array:

```
>>> a_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [1, 2, 3, 4]])
```

You can find unique values with:

```
>>> unique_values = np.unique(a_2d)
>>> print(unique_values)
[ 1 2 3 4 5 6 7 8 9 10 11 12]
```

If the axis argument isn't passed, your 2D array will be flattened.

If you want to get the unique rows or columns, make sure to pass the axis argument. To find the unique rows, specify axis=0 and for columns, specify axis=1.

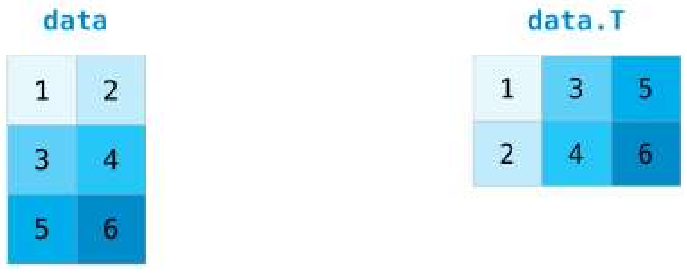
```
>>> unique_rows = np.unique(a_2d, axis=0)
>>> print(unique_rows)
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]]
```

To get the unique rows, index position, and occurrence count, you can use:

```
>>> unique_rows, indices, occurrence_count = np.unique(
... a_2d, axis=0, return_counts=True, return_index=True)
>>> print(unique_rows)
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]]
>>> print(indices)
[0 1 2]
>>> print(occurrence_count)
[2 1 1]
```

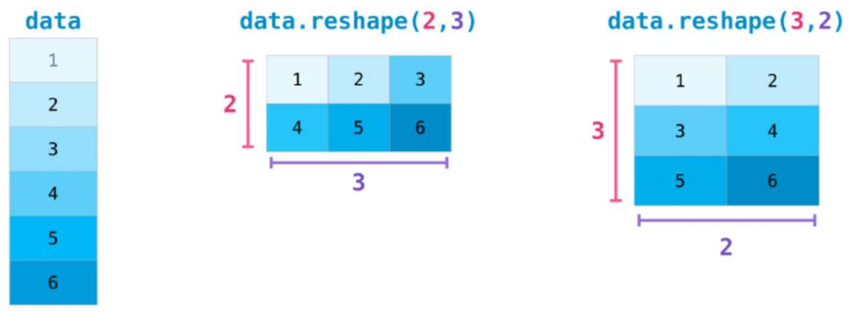
Transposing and reshaping a matrix

It's common to need to transpose your matrices. NumPy arrays have the property T that allows you to transpose a matrix.



You may also need to switch the dimensions of a matrix. This can happen when, for example, you have a model that expects a certain input shape that is different from your dataset. This is where the reshape method can be useful. You simply need to pass in the new dimensions that you want for the matrix.

```
>>> data.reshape(2, 3)
array([[1, 2, 3],
          [4, 5, 6]])
>>> data.reshape(3, 2)
array([[1, 2],
          [3, 4],
          [5, 6]])
```



You can also use .transpose() to reverse or change the axes of an array according to the values you specify.

If you start with this array:

```
>>> arr = np.arange(6).reshape((2, 3))
>>> arr
array([[0, 1, 2],
[3, 4, 5]])


>>> arr.transpose()
array([[0, 3],
          [1, 4],
          [2, 5]])
```

You can also use arr.T:

```
>>> arr.T
array([[0, 3],
          [1, 4],
          [2, 5]])
```

Reverse an array

NumPy’s np.flip() function allows you to flip, or reverse, the contents of an array along an axis. When using np.flip(), specify the array you would like to reverse and the axis. If you don’t specify the axis, NumPy will reverse the contents along all of the axes of your input array.

Reversing a 1D array

If you begin with a 1D array like this one:

```
>>> arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

You can reverse it with:

```
>>> reversed_arr = np.flip(arr)
```

If you want to print your reversed array, you can run:

```
>>> print('Reversed Array: ', reversed_arr)
Reversed Array: [8 7 6 5 4 3 2 1]
```

Reversing a 2D array

A 2D array works much the same way.

If you start with this array:

```
>>> arr_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can reverse the content in all of the rows and all of the columns with:

```
>>> reversed_arr = np.flip(arr_2d)
>>> print(reversed_arr)
[[12 11 10 9]
[ 8 7 6 5]
[ 4 3 2 1]]
```

You can easily reverse only the *rows* with:

```
>>> reversed_arr_rows = np.flip(arr_2d, axis=0)
>>> print(reversed_arr_rows)
[[ 9 10 11 12]
[ 5 6 7 8]
[ 1 2 3 4]]
```

Or reverse only the *columns* with:

```
>>> reversed_arr_columns = np.flip(arr_2d, axis=1)
>>> print(reversed_arr_columns)
[[ 4 3 2 1]
[ 8 7 6 5]
[12 11 10 9]]
```

You can also reverse the contents of only one column or row. For example, you can reverse the contents of the row at index position 1 (the second row):

```
>>> arr_2d[1] = np.flip(arr_2d[1])
>>> print(arr_2d)
[[ 1 2 3 4]
[ 8 7 6 5]
[ 9 10 11 12]]
```

You can also reverse the column at index position 1 (the second column):

```
>>> arr_2d[:,1] = np.flip(arr_2d[:,1])
>>> print(arr_2d)
[[ 1 10 3 4]
[ 8 7 6 5]
[ 9 2 11 12]]
```

Reshaping and flattening multidimensional arrays

There are two popular ways to flatten an array: .flatten() and .ravel(). The primary difference between the two is that the new array created using ravel() is actually a reference to the parent array (i.e., a “view”). This means that any changes to the new array will affect the parent array as well. Since ravel does not create a copy, it’s memory efficient.

If you start with this array:

```
>>> x = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use flatten to flatten your array into a 1D array.

```
>>> x.flatten()
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

When you use flatten, changes to your new array won’t change the parent array.

For example:

```
>>> a1 = x.flatten()
>>> a1[0] = 99
>>> print(x) # Original array
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]]
>>> print(a1) # New array
[99 2 3 4 5 6 7 8 9 10 11 12]
```

But when you use ravel, the changes you make to the new array will affect the parent array.

For example:

```
>>> a2 = x.ravel()
>>> a2[0] = 98
>>> print(x) # Original array
[[98 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]]
>>> print(a2) # New array
[98 2 3 4 5 6 7 8 9 10 11 12]
```

NumPy Array Searching (Search from arrays)

You can search an array for a certain value and return the indexes that get a match.

To search an array, use the ***where()*** method.

Example

Find the indexes where the value is 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Output

It will return a tuple:

```
(array([3, 5, 6],)
```

Which means that the value 4 is present at index 3, 5, and 6.

Example

Find the indexes where the values are even:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

Output

```
(array([1, 3, 5, 7]),)
```

Search Sorted

There is a method called ***searchsorted()*** which performs a binary search in the array and returns the index where the specified value would be inserted to maintain the search order.

The ***searchsorted()*** method is assumed to be used on sorted arrays.

Example

Find the indexes where the value 7 should be inserted:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

Output

```
1
```

Example explained: The number 7 should be inserted on index 1 to remain the sort order.

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

Search from the Right Side

By default, the left most index is returned, but we can give ***side='right'*** to return the right most index instead.

Example

Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

Output

```
2
```

Example explained: The number 7 should be inserted on index 2 to remain the sort order.

The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

Multiple Values

To search for more than one value, use an array with the specified values.

Example

Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np
arr = np.array([1, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 6])
print(x)
```

Output

```
[1 2 3]
```

The return value is an array: [1 2 3] containing the three indexes, where 2, 4, 6 would be inserted in the original array to maintain the order.

Mathematical Functions

Trigonometric Functions:

Example

```
import numpy as np

adegrees = np.arange(0, 360, 30)  # here angles are in degrees
##adegrees = np.array([0,30,45,60,90]) # here angles are in degrees

# Convert angle from degrees to radians by multiplying with pi/180
aradians = np.pi * adegrees / 180

sin = np.sin(aradians)
print('Sine of different angles:’)
print(sin)

cos = np.cos(aradians)
print('Cosine of different angles:’)
print(cos)

tan = np.tan(aradians)
print('Tangent of different angles:’)
print(tan)
```

arcsin, ***arccos***, and ***arctan*** functions return the trigonometric inverse of **sin**, **cos**, and **tan** respectively of the given angle.

The ***numpy.degrees()*** **function** converts radians to degrees.

Functions for Rounding

numpy.around()

This is a function that returns the value rounded to the desired precision. The function takes the following parameters.

numpy.around(a, decimals)

Where,

Sr.No.	Parameter & Description
1	**a** Input data

2	**decimals** The number of decimals to round to. Default is 0. If negative, the integer is rounded to position to the left of the decimal point

Example

```
import numpy as np
a = np.array([1.0,5.55, 123, 0.567, 25.532])

print('Original array:')
print(a)

print('After rounding:')
print(np.around(a))
print(np.around(a, decimals = 1))
print(np.around(a, decimals = -1))
```

numpy.floor()

This function returns the largest integer less than or equal to the input parameter. The floor of the **scalar x** is the largest **integer i**, such that **i** <= **x**. Note that in Python, flooring is always rounded away from 0.

Example

```
import numpy as np

a = np.array([-1.7, 1.5, -0.2, 0.6, 10])
print('The given array:’)
print(a)
print('The modified array:’)
print(np.floor(a))
```

numpy.ceil()

This function returns the smallest integer greater than or equal to the input parameter. The ceil of the **scalar x** is the smallest **integer i**, such that **i** >= **x**.

Example

```
import numpy as np

a = np.array([-1.7, 1.5, -0.2, 0.6, 10])
print('The given array:’)
print(a)
print('The modified array:')
print(np.ceil(a))
```

Statistical Functions

NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc. from the given elements in the array. The functions are explained as follows –

numpy.amin() *and* ***numpy.amax()***

These functions return the minimum and the maximum from the elements in the given array along the specified axis.

Example

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])

print( 'Our array is:')
print( a)
print( '\n')

print( 'Applying amin() function:')
print( np.amin(a,1) )
print( '\n')

print( 'Applying amin() function again:')
print( np.amin(a,0) )
print( '\n')

print( 'Applying amax() function:' )
print( np.amax(a) )
print( '\n' )

print( 'Applying amax() function again:' )
print( np.amax(a, axis = 0))
```

It will produce the following output –

```
Our array is:
[[3 7 5]
[8 4 3]
[2 4 9]]

Applying amin() function:
[3 3 2]

Applying amin() function again:
[2 4 3]
```

```
Applying amax() function:
9

Applying amax() function again:
[8 7 9]
```

numpy.percentile()

Percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. The function ***numpy.percentile()*** takes the following arguments.

numpy.percentile(a, q, axis)

where,

Sr.No.	Argument & Description
1	**a** Input array
2	**q** The percentile to compute must be between 0-100
3	**axis** The axis along which the percentile is to be calculated

Example

```
import numpy as np
a = np.array([[30,40,70],[80,20,10],[50,90,60]])

print( 'Our array is:' )
print( a )
print( '\n')

print( 'Applying percentile() function:')
print( np.percentile(a,50) )
print( '\n')

print( 'Applying percentile() function along axis 1:' )
print( np.percentile(a,50, axis = 1) )
print( '\n' )

print( 'Applying percentile() function along axis 0:' )
print( np.percentile(a,50, axis = 0))
```

It will produce the following output −

```
Our array is:
[[30 40 70]
 [80 20 10]
 [50 90 60]]

Applying percentile() function:
50.0

Applying percentile() function along axis 1:
[ 40. 20. 60.]

Applying percentile() function along axis 0:
[ 50. 40. 60.]
```

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half. The ***numpy.median()*** function is used as shown in the following program.

Example

```
import numpy as np
a = np.array([[30,65,70],[80,95,10],[50,90,60]])

print( 'Our array is:' )
print( a )
print( '\n')

print( 'Applying median() function:' )
print( np.median(a) )
print( '\n' )

print( 'Applying median() function along axis 0:' )
print( np.median(a, axis = 0) )
print( '\n' )

print( 'Applying median() function along axis 1:' )
print( np.median(a, axis = 1))
```

It will produce the following output –

```
Our array is:
[[30 65 70]
 [80 95 10]
 [50 90 60]]

Applying median() function:
```

```
65.0

Applying median() function along axis 0:
[ 50. 90. 60.]

Applying median() function along axis 1:
[ 65. 80. 60.]
```

numpy.mean()

Arithmetic mean is the sum of elements along an axis divided by the number of elements. The ***numpy.mean()*** function returns the arithmetic mean of elements in the array. If the axis is mentioned, it is calculated along it.

Example

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])

print( 'Our array is:' )
print( a )
print( '\n')

print( 'Applying mean() function:' )
print( np.mean(a) )
print( '\n' )

print( 'Applying mean() function along axis 0:')
print( np.mean(a, axis = 0) )
print( '\n' )

print( 'Applying mean() function along axis 1:')
print( np.mean(a, axis = 1))
```

It will produce the following output –

```
Our array is:
[[1 2 3]
 [3 4 5]
 [4 5 6]]

Applying mean() function:
3.66666666667

Applying mean() function along axis 0:
[ 2.66666667 3.66666667 4.66666667]

Applying mean() function along axis 1:
[ 2. 4. 5.]
```

numpy.average()

Weighted average is an average resulting from the multiplication of each component by a factor reflecting its importance. The ***numpy.average()*** function computes the weighted average of elements in an array according to their respective weight given in another array. The function can have an axis parameter. If the axis is not specified, the array is flattened.

Considering an array [1,2,3,4] and corresponding weights [4,3,2,1], the weighted average is calculated by adding the product of the corresponding elements and dividing the sum by the sum of weights.

Weighted average = (1*4+2*3+3*2+4*1)/(4+3+2+1)

Example

```
import numpy as np
a = np.array([1,2,3,4])

print( 'Our array is:' )
print( a )
print( '\n' )

print( 'Applying average() function:' )
print( np.average(a) )
print( '\n' )

# this is same as mean when weight is not specified
wts = np.array([4,3,2,1])

print( 'Applying average() function again:' )
print( np.average(a,weights = wts) )
print( '\n' )

# Returns the sum of weights, if the returned parameter is set to True.
print( 'Sum of weights'
print( np.average([1,2,3, 4],weights = [4,3,2,1], returned = True))
```

It will produce the following output –

```
Our array is:
[1 2 3 4]

Applying average() function:
2.5

Applying average() function again:
2.0

Sum of weights
(2.0, 10.0)
```

In a multi-dimensional array, the axis for computation can be specified.

Example

```
import numpy as np
a = np.arange(6).reshape(3,2)

print( 'Our array is:' )
print( a )
print( '\n' )

print( 'Modified array:' ))
wt = np.array([3,5])
print( np.average(a, axis = 1, weights = wt) )
print( '\n' )

print( 'Modified array:' )
print( np.average(a, axis = 1, weights = wt, returned = True))
```

It will produce the following output –

```
Our array is:
[[0 1]
 [2 3]
 [4 5]]

Modified array:
[ 0.625 2.625 4.625]

Modified array:
(array([ 0.625, 2.625, 4.625]), array([ 8., 8., 8.]))
```

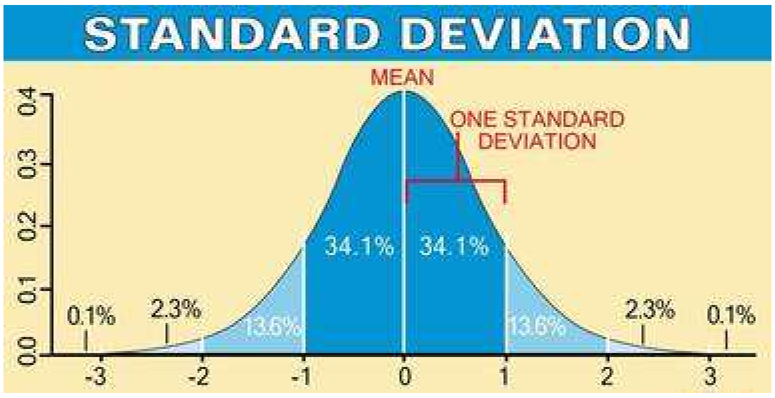
Standard Deviation

Standard deviation is the square root of the average of squared deviations from mean. The formula for standard deviation is as follows –

std = sqrt(mean(abs(x - x.mean())**2))

Standard deviation is a statistical measurement of the amount a number varies from the average number in a series. A low standard deviation means that the data is very closely related to the average, thus very reliable. A high standard deviation means that there is a large variance between the data and the statistical average and is not as reliable.

Standard deviation is a measure of how spread out a data set is. It's used in a huge number of applications. In finance, standard deviations of price data are frequently used as a measure of volatility. In opinion polling, standard deviations are a key part of calculating margins of error.



If the array is [1, 2, 3, 4], then its mean is 2.5. Hence the squared deviations are [2.25, 0.25, 0.25, 2.25] and the square root of its mean divided by 4, i.e., sqrt (5/4) is 1.1180339887498949.

Example

```
import numpy as np
print (np.std([1,2,3,4]))
```

It will produce the following output −

```
1.1180339887498949
```

Variance

Variance is the average of squared deviations, i.e., ***mean(abs(x - x.mean())**2)***. In other words, the standard deviation is the square root of variance.

In statistics, variance measures variability from the average or mean. A large variance indicates that numbers in the set are far from the mean and far from each other. A small variance, on the other hand, indicates the opposite. A variance value of zero, though, indicates that all values within a set of numbers are identical. Every variance that isn't zero is a positive number. A variance cannot be negative. That's because it's mathematically impossible since you can't have a negative value resulting from a square. Variance is an important metric in the investment world. Variability is volatility, and volatility is a measure of risk. It helps assess the risk that investors assume when they buy a specific asset and helps them determine whether the investment will be profitable.

Example

```
import numpy as np
print (np.var([1,2,3,4]))
```

It will produce the following output −

```
1.25
```