**Python 3**

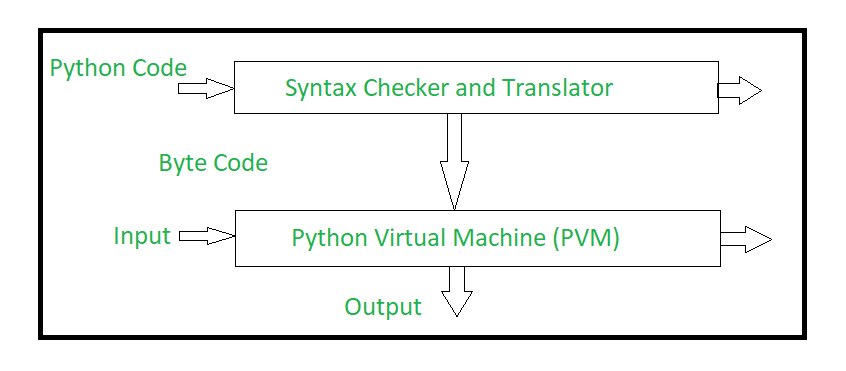
**Python** is an object-oriented programming language like Java. Python is called an interpreted language.

The standard implementation of python is called “cpython”. It is the default and widely used implementation of Python.

Python doesn’t convert its code into machine code, something that hardware can understand. It actually converts it into something called byte code. So within python, compilation happens, but it’s just not into a machine language. It is into byte code (.pyc or .pyo) and this byte code can’t be understood by the CPU. So, we need an interpreter called the python virtual machine to execute the byte codes.

# Internal Architecture for Python

## PVM



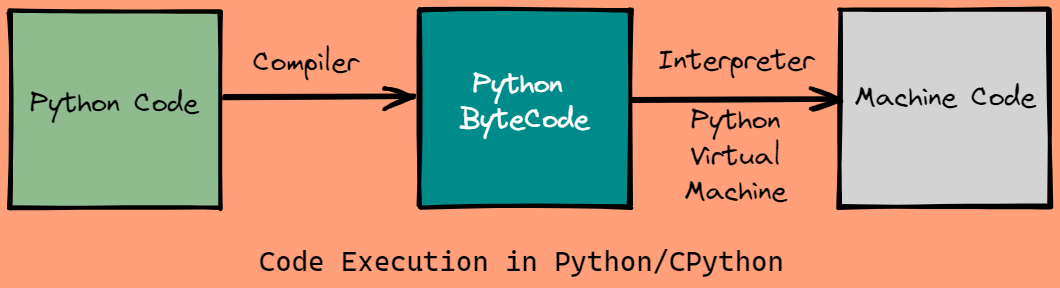
* **Step 1:** The python compiler reads a python source code or instruction. Then it verifies that the instruction is well-formatted, i.e. it checks the syntax of each line. If it encounters an error, it immediately halts the translation and shows an error message.
* **Step 2:**If there is no error, i.e. if the python instruction or source code is well-formatted then the compiler translates it into its equivalent form in an intermediate language called “Byte code”.
* **Step 3:**Byte code is then sent to the Python Virtual Machine(PVM) which is the python interpreter. PVM converts the python byte code into machine-executable code. If an error occurs during this interpretation, then the conversion is halted with an error message.

## Comparison between other implementations of Python

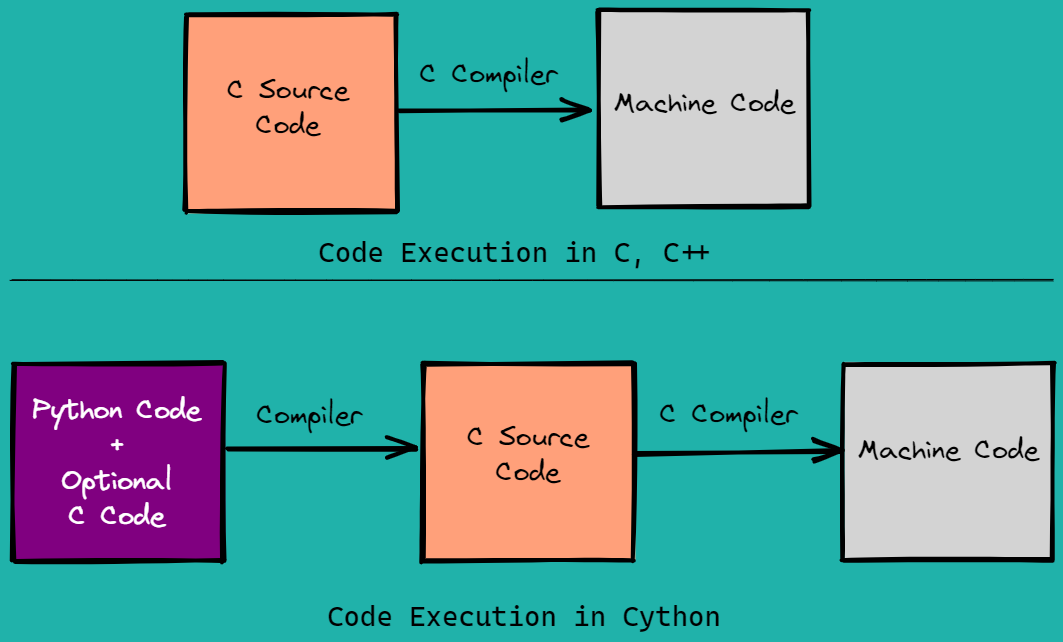
As *python is open source that's why we can customize python as per our requirements*. After customization, we can name that version as we want. That's why multiple flavours of python are available. Each flavour is a customized version of python to fulfill a special requirement. It is similar to the fact that there are multiple flavours of UNIX like, Ubuntu, Linux, RedHat Linux etc. Below are some of the flavours of python :

1. CPython - **Default implementation of python** programming language which we download from python.org, provided by python software foundation. It is *written in C and python*. It **does not allow us to write any C code, only python code** are allowed. CPython can be called as *both an interpreter and a compiler* as here our *python code first gets compiled to python bytecode* then the *bytecode gets interpreted into machine code by PVM or Python Virtual Machine*.

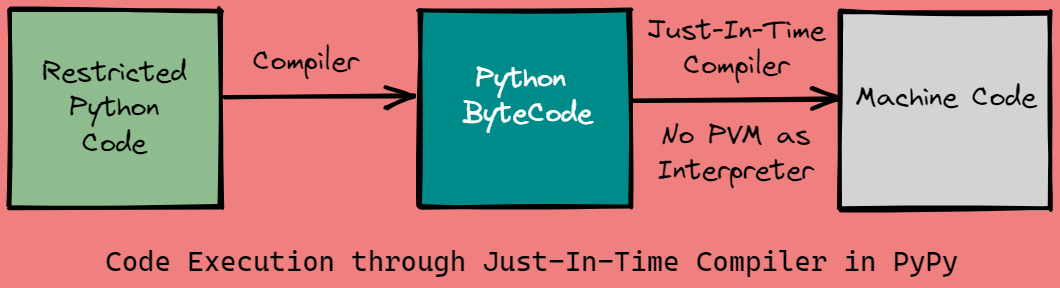
**Only JavaScript is a fully interpreted language** as JavaScript source code directly gets interpreted one by one line into machine code, no bytecode involved. A JavaScript engine can be implemented as a standard interpreter or Just-In-Time Compiler, like Google Chrome's V8 engine, no bytecode involved, but stores already compiled machine code based on decisions like how often a piece of code is executed etc.

[](https://i.stack.imgur.com/UEPNg.png)

1. Cython - Cython is a **programming language which is a superset of python and C**. It is *written in C and python*. It is designed to give **C-like performance with python syntax and optional C-syntax**. Cython is a *compiled language as it generates C code and gets compiled by C compiler*. We can write *similar code in Cython as in default python or CPython*, the **differences** are :
   * *Cython allows* us to write *optional additional C code* and,
   * *In Cython*, our *python code gets translated into C code internally* so that it can get compiled by C compiler. Although Cython results in considerably **faster execution, but falls short of original C** language execution. This is *because Cython has to make calls to the CPython interpreter and CPython standard libraries* to understand the written CPython code

[](https://i.stack.imgur.com/WwhR2.png)

1. JPython / Jython - **Java implementation of python** programming language. It is *written in Java and python*. Here our *python code first gets compiled to Java bytecode* and then *that bytecode gets interpreted to machine code by JVM* or Java Virtual Machine. This is similar to how Java code gets executed : Java code first gets compiled to intermediate Bytecode and then that Bytecode gets interpreted to machine code by JVM
2. PyPy - **RPython implementation of python** programming language. It is *written in a restricted subset of python called Restricted Python (RPython)*. *PyPy runs faster than CPython* because **to interpret bytecode, PyPy has a Just-in-time Compiler while CPython has an Interpreter**.

[](https://i.stack.imgur.com/5yFZe.png)

*Compiler translates our high level source code into bytecode* and *to translate bytecode into machine code, some implementations have normal interpreter, some have Just-in-time compiler*. To execute a loop which runs say, million times, a **JIT compiler translates bytecode into machine code only once and for next iterations machine just understands the bytecode**. *JIT Compiler stores already compiled machine code based on decisions like how often a piece of code is executed* etc., so it does not have to translate bytecode into machine code each time. Drawback of JIT Compiler is initial slower execution when the code gets analysed. Whereas *normal Interpreter*, in *each iteration*, *repeatedly translates bytecode into machine code* thereby taking more time to complete a loop which runs say, million times

* IronPython - **C# implementation of python**, targeting the .NET framework
* Ruby Python - works with Ruby platform
* Anaconda Python - **Distribution of python and R programming languages for scientific computing** like, *data science, machine learning, artificial intelligence, deep learning, handling large volume of data* etc. Numerous number of *libraries like, scikit-learn, tensorflow, pytorch, numba, pandas, jupyter, numpy, matplotlib etc.* are available with this package
* Stackless - **Python for Concurrency**

To test speed of each implementation, we write a program to call integrate\_f 500 times using an N value of 50,000, and record the execution time over several runs. Below table shows the benchmark results :

| **Implementation** | **Execution Time (seconds)** | **Speed Up** |
| --- | --- | --- |
| CPython | 9.25 |  |
| *CPython + Cython* | *0.21* | *44x* |
| PyPy | *0.57* | *16x* |

<https://stackoverflow.com/questions/17130975/python-vs-cpython>

## Python vs Java

|  |  |
| --- | --- |
| **Python** | **Java** |
| Python is an Interpreted Language | Java is a Compiled Language |
| Module | Packages |
| 100 % Object Oriented | Can be 100 % Oriented |
| Functions | Methods |
| Latest 3.XX (3.10) | JDK 17 LTS., JDK 18 |
| Always Dynamically Typed | Statically Typed (But we can achieve Dynamic Typing using var keyword JDK 11 |
| Slow in Execution since its interpreted | Fast in Execution Since its compiled |

## PEP-8

Pep-8 is a coding standard Guidelines or Python

<https://peps.python.org/pep-0008/#:~:text=Modules%20should%20have%20short%2C%20all,use%20of%20underscores%20is%20discouraged>.

# Basics

## Hello World

**hello.py**

print('Hello World')

print("Hello World")

print('''Hello World''')

print('''"Hello World"''')

## Basic Data Type

**basic\_data\_types.py**

# str

simran ="Simran Bhat"

print(len(simran)) #11

print((type(simran))) #<class 'str'>

# float

a =8.9

print((type(a))) #<class 'float'>

# int

b = 8

print(type(b)) #<class 'int'>

c= a+b

print(type(c)) #<class 'float'>

# bool

d = True

print(d)

print(type(d)) #<class 'bool'>

b = d #this is possible in Python as everything is object

print(b) #<class 'bool'>

x = 7 + 8j

print(type(x)) #<class 'complex'>

# Examples

first\_character = 'X'

print(type(first\_character))#<class 'str'> as there is not char data type in python

e = 10

print(type(e)) #<class 'int'>

#There is no physical limit for int data type

e = 10000000000000000000000000000000000000000000

print(type(e)) #<class 'int'>

## Command Line Arguments

**command\_line\_args.py**

import sys

# To use Command Line Arguments, we use an inbuilt module called sys

print("Hello")

print(sys.argv[0])

#CommandLineArguments.py (the 1st parameter will always be the file name)

print(sys.argv[1])

#Python

print(sys.argv[2])

#is

print(sys.argv[-1])

#with

print(type(sys)) #<class 'module'>

print(type(sys.argv)) #<class 'list'>

print(len(sys.argv)) #9

***execution…***

suman@MrHour MINGW64 ~/Downloads/code/Python/Recall\_PY/Basics

$ **python** command\_line\_args.py Python *is a good language to begin with*

Hello

command\_line\_args.py

Python

is

with

<class 'module'>

<class 'list'>

9

## Introduction to Special Variables in Python

### Why is the \_ \_name\_ \_ variable used?

The \_\_name\_\_ variable (two underscores before and after) is a special Python variable. It gets its value depending on how we execute the containing script.

Sometimes you write a script with functions that might be useful in other scripts as well. In Python, you can import that script as a module in another script.

Thanks to this special variable, you can decide whether you want to run the script. Or that you want to import the functions defined in the script.

<https://www.freecodecamp.org/news/whats-in-a-python-s-name-506262fe61e8/#:~:text=The%20__name__%20variable%20(two%20underscores%20before%20and%20after,a%20module%20in%20another%20script>.

**sum\_of\_two\_numbers.py**

# Global Variables

a=20

b=80

product=a\*b

power = a\*\*2

# print(sum) #100

# print(power) #400

# def keyword is used to define functions in python

def add(a,b):

# local variables

c = 9

print(\_\_name\_\_)

#\_\_main\_\_

#Stores the name of current file(module)

print(a+b)

#print(\_\_name\_\_)#\_\_main\_\_

#The below line will prevent from automatic execution while importing in other modules

if \_\_name\_\_ == "\_\_main\_\_":

add(5,5)

print(f' the evaluation of {a}^2 is {power}') #the evaluation of a=20^2 is 400

print(type(\_\_name\_\_)) #<class 'str'>

print(\_\_name\_\_) #\_\_main\_\_

**special\_keywords\_in\_python.py**

import sum\_of\_two\_numbers as simy

# in python every file is a module

if \_\_name\_\_ == "\_\_main\_\_":

    simy.add(5,5) # sum\_of\_two\_numbers 10

    print(simy.a) #20

    print(type(\_\_name\_\_))#\_\_main\_\_

    print(\_\_name\_\_)#\_\_main\_\_

## if-elif-else using turtle example

**turtle\_graphics.py**

import turtle

def square(length):

    # default orientation is right

    rafale=turtle.Turtle()

    print(type(rafale))

    rafale.fd(length)

    rafale.rt(90)

    rafale.fd(length)

    rafale.rt(90)

    rafale.fd(length)

    rafale.rt(90)

    rafale.fd(length)

    rafale.rt(90)

def square\_using\_forloop(length):

    # default orientation is right

    rafale=turtle.Turtle()

    for i in range(4):

        print(i)

        rafale.fd(length)

        rafale.rt(90)

def polygon\_of\_n\_sides(n,length):

    if(n<0):

        print(f'Illegal closed shape of sides {n}')

    elif(n<3):

        print(f"Polygon impossible with sides {n}")

        return

    else:

        angle\_of\_turn = 360/n

        rafale = turtle.Turtle()

        for i in range(n):

            rafale.fd(length)

            rafale.lt(angle\_of\_turn)

if \_\_name\_\_ == "\_\_main\_\_":

    print(type(turtle))

    # square\_using\_forloop(300)

    polygon\_of\_n\_sides(length=2,n=100)#keyword arguments

    # square(400)

## Various arguments passed to a function

**fn\_args\_types.py**

def power(base, exponent=1):#default arguments

    print(base\*\*exponent)

def show\_student\_data (\*details):

    print(type(details))# tuple is the type for var args

    for i in details:

        print(i,end=" ")#end =" " parameter prevents changing of line while printing

        #and adds a space between each printing operation

def show\_details\_student (\*\*details):

    print(type(details))# dictionary

    for key,value in details.items():

        print(f'STUDENT ROLL: {key} | STUDENT NAME: {value} ')

if \_\_name\_\_ == "\_\_main\_\_":

    show\_student\_data("Simarn", 1,"Suman",89)#vargs

    print()#to change the line

    list\_of\_data = ["Simarn", 1,"Suman",89]

    show\_student\_data(\*list\_of\_data)#dynamic unpacking

    print()#to change the line

    #kwargs

    # details\_of\_student = {1:"Suman" , 2 :"Amy" , 8 :'Simran', 3:"Satya"}#dictionary

    # show\_details\_student(first ='Python', mid ='is', last='Fun')#dynamic unpacking

    power(2)#default args (exponent=1)

    power(2,10)#positional args

    power(exponent=10, base=2)#keyword args

## Loops in Python

**while\_loop.py**

def swap\_a\_number(a,b):

c = a

a = b

b = c

return a,b # returning multiple arguments in form of tuple

#dynamic swapping

def swap\_a\_number\_dynamic(a,b):

print(f'BEFORE : a = {a} b = {b}')

a , b = b , a#Dynamic Internal Pythn swap

print(f'AFTER : a = {a} b = {b}')

# while loop

def factorial(n):

result = 1

while(n>1):

result \*=n

n-=1

return result

#for loop and range fn

def play\_with\_range():

for i in range(-1,11):#start and stop-1

print(i,end=" ")

print()

for i in range(-1,11,2):#start, stop-1 and steps

print(i,end=" ")

print()

for i in range(11,-4,-2):#start, stop-1 and reverse steps

print(i,end=" ")

print()

if \_\_name\_\_ == "\_\_main\_\_":

ans=factorial(10)

print(ans)

play\_with\_range()

a = 8

b = 9

swapped\_number = swap\_a\_number(a,b)

print(id(swapped\_number))#prints unique ID of the object

print(type(swapped\_number))#<class 'tuple'>

print(swapped\_number)

a,b = swapped\_number#dynamic unpacking of tuple

print(f'a = {a} b = {b}')

swap\_a\_number\_dynamic(a,b)