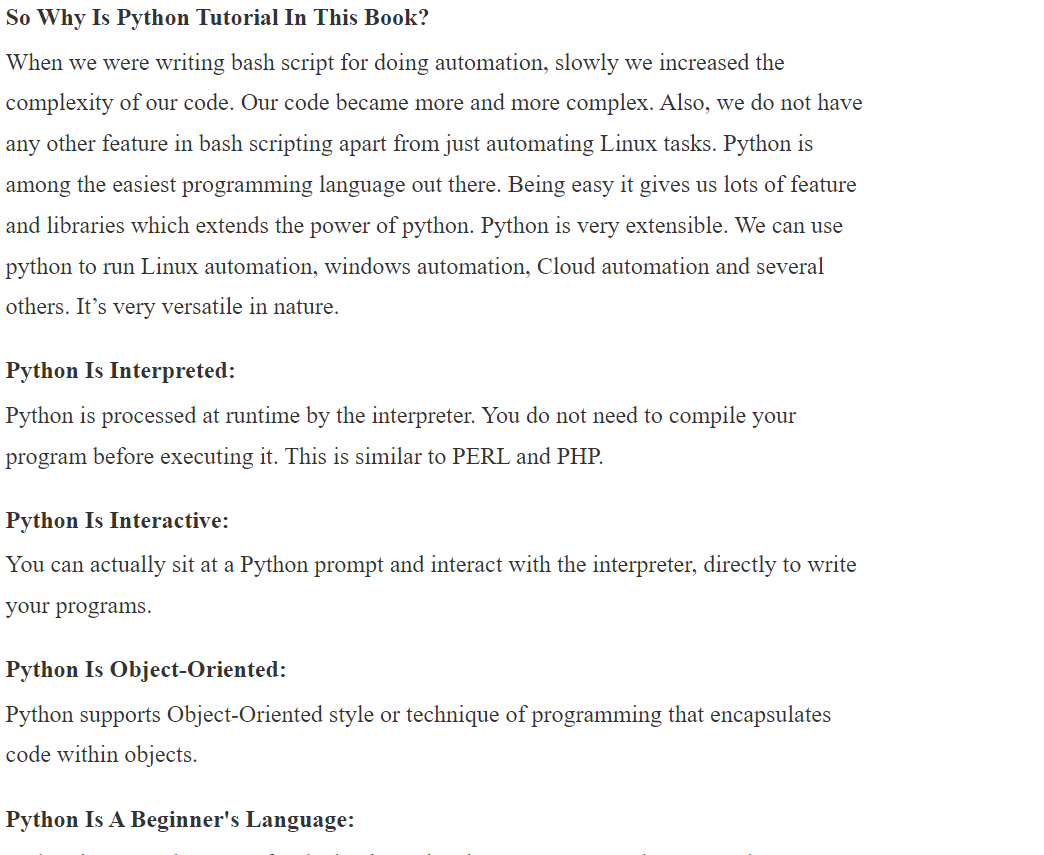
# Python

**Already I have knowledge concept of python before but this doc refer is based on basic devops point of view**

🡪 it used for really complex automation frameworks or pipeline where the automation tools like Jenkins, ansible, terraform cannot help.

* Compare to programing language automation tools is more sophisticated and easily to manage



In Linux machine python in installed default because some of the packages like **yum** etc are build with python

Some of different in python2 and python3

python

1. vim python3\_basic.py

#!/usr/bin/python3

print("this my first python program in centos machine")

1. chmod +x python3\_basic.py
2. ./python3\_basic.py
3. history

#!/bin/bash

x=0

echo "learning indentation"

echo

if [ $x -eq 0 ]

then

echo "if block is executed"

echo "value equal-to 0"

else

echo "in the else block"

echo "value of x is non zero"

fi

echo

echo "out of if/else blocks"

or

x=0

print("learning identation ")

if(x == 0):

print("if block is executing")

print("value is equeal to zero")

else:

print("else block is executing")

print("value is not equal to zero ")

print("out of if/else block")

# COMMENTS AND QUOTES

# this is”#” single line comment

Multiline comments ‘’’ ‘’’

**‘’’ this is muitiline comment**

**------------------------**

**----------------------** ‘’’

Assign value

Skill = “devops” we can assign the string value in single quotes and double quotes

Print(“devops”)

Print(‘devops’)

Print(skill)

**We can print paragraph like this**

Print(‘’’ this is a parapraph

-----------------------

------------------------ ‘’’)

# Variables

Variable assignments

Var1 = ‘python’

Var2 = 75

Var3 = 3.5

n Python, variables are used to store and manipulate data. They serve as labels or names that reference values stored in memory. Python is dynamically typed, which means you don't need to declare the data type of a variable explicitly; Python infers the type based on the value assigned to it. Here's a more advanced explanation of variables in Python:

1. Variable Assignment: To create a variable, you simply assign a value to a name using the assignment operator (**=**). Python will automatically determine the data type of the variable based on the value:

x = 10 # x is an integer

y = 3.14 # y is a float

name = "John" # name is a string

1. Variable Naming Conventions: Variable names in Python should follow these conventions:
   * Start with a letter or underscore (\_).
   * Can contain letters, numbers, and underscores.
   * Case-sensitive (e.g., **myVar** and **myvar** are different).
   * Avoid using reserved words (e.g., **if**, **while**, **for**) as variable names.
2. Variable Reassignment: You can change the value of a variable by assigning it a new value:

X == 20

1. Multiple Assignment: Python allows multiple variables to be assigned values in a single line:

a, b, c = 1, 2, 3

1. Variable Types: Python has several built-in data types, including integers, floats, strings, lists, tuples, sets, dictionaries, and custom objects (classes). Variables can be assigned values of any of these types.
2. Variable Scope: The scope of a variable defines where it can be accessed or modified in your code. Variables can be local (inside a function or block), global (defined outside functions), or nonlocal (used in nested functions). Understanding variable scope is crucial for writing maintainable code.

global\_var = 10

def my\_function():

local\_var = 5 # Local variable

print(global\_var) # Access global variable

my\_function()

print(local\_var) # Raises NameError, as local\_var is not accessible here

1. Dynamic Typing: Python is dynamically typed, meaning that variable types are determined at runtime. You can change the type of a variable by assigning it a different value:

x = 5

x = "Hello"

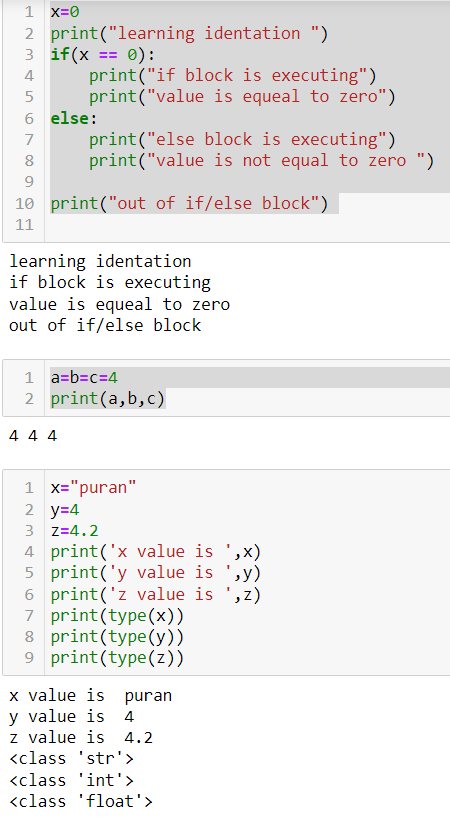
1. Type Annotations (Optional): While Python is dynamically typed, you can use type annotations with variable names to indicate the expected data type. This is especially useful for documenting your code and for static analysis tools like linters and type checkers:

age: int = 30

name: str = "Alice"

1. Memory Management: Python handles memory management, including garbage collection, automatically. You don't need to worry about allocating or deallocating memory for variables.
2. Variable Naming Style: Following Python's naming conventions (PEP 8) is recommended for readability. Use lowercase letters and underscores for variable names (e.g., **my\_variable\_name**).

In summary, Python variables are dynamically typed, flexible, and versatile tools for working with data. Understanding variable scope, naming conventions, and type annotations can help you write clean, maintainable code.



In Python, lists, sets, and tuples are three different data structures, each with its own characteristics and use cases. Here's a brief overview of each with examples:

1. Lists:
   * Lists are ordered collections of elements.
   * They are mutable, which means you can change their content (add, remove, or modify elements).
   * Lists are defined using square brackets **[]**.

my\_list = [1, 2, 3, 4, 5]

my\_list.append(6) # Add an element to the end

my\_list[2] = 7 # Modify an element

my\_list.remove(3) # Remove an element by value

2) Sets:

* Sets are unordered collections of unique elements.
* They are mutable, which means you can add or remove elements.
* Sets are defined using curly braces **{}** or the **set()** constructor.

my\_set = {1, 2, 3, 4, 5}

my\_set.add(6) # Add an element

my\_set.remove(3) # Remove an element by value

**Note: Sets automatically remove duplicate values, so if you try to add a duplicate element, it won't change the set.**

Tuples:

* Tuples are ordered collections of elements.
* They are immutable, which means you cannot change their content once they are created.
* Tuples are defined using parentheses **()** or simply by separating values with commas.

Example:

**my\_tuple = (1, 2, 3, 4, 5)**

**# Accessing elements in a tuple**

**first\_element = my\_tuple[0]**

**second\_element = my\_tuple[1]**

1. Attempting to modify a tuple will result in an error.

Here are the key differences between these data structures:

* Lists are ordered and allow duplicate elements, while sets are unordered and contain only unique elements.
* Lists and sets are mutable, but tuples are immutable.
* Lists use square brackets **[]**, sets use curly braces **{}**, and tuples use parentheses **()**.

Choose the appropriate data structure based on your specific requirements. If you need an ordered collection that allows duplicates and can be modified, use a list. If you need a collection of unique, unordered elements that can be modified, use a set. If you need an ordered, immutable collection, use a tuple.

**Dictionaries**: A dictionary is a collection of key-value pairs, where each key is unique. Dictionaries are enclosed in curly braces **{}** and can be used to store and retrieve data based on keys.

Example:

**# Creating a dictionary**

**student = {**

**"name": "Alice",**

**"age": 20,**

**"grade": "A"**

**}**

**# Accessing values**

**print(student["name"]) # Output: "Alice"**

**# Modifying values**

**student["age"] = 21**

**# Adding a new key-value pair**

**student["city"] = "New York"**

**# Iterating through the dictionary**

**for key, value in student.items():**

**print(key, ":", value)**

**Custom Objects (Classes)**: You can define custom classes to create your own objects with attributes and methods. Here's a simple example:

**class Dog:**

**def \_\_init\_\_(self, name, age):**

**self.name = name**

**self.age = age**

**def bark(self):**

**print(f"{self.name} says Woof!")**

**# Creating objects**

**dog1 = Dog("Buddy", 3)**

**dog2 = Dog("Rex", 5)**

**# Accessing attributes and calling methods**

**print(dog1.name) # Output: "Buddy"**

**dog2.bark() # Output: "Rex says Woof!"**

**Booleans**: Booleans represent either **True** or **False**. They are often used for conditional statements and logical operations.

Example:

**x = 5**

**y = 10**

**is\_greater = x > y # This will be False**

**is\_equal = x == 5 # This will be True**

**if is\_greater:**

**print("x is greater than y")**

**elif is\_equal:**

**print("x is equal to 5")**

**else:**

**print("x is neither greater than y nor equal to 5")**

# print format

name= "sars19"

virus = "corona"

print("the name of the virus is :",name)

print("the name of the virus is {}.".format(name))

print("the name of the virus is {} and descrise name {}.".format(name,virus))

print(f'the name of the viurs is {name}and descrise name {virus}.')

print("the name of the virus is :"+" "+name )

output:

the name of the virus is : sars19

the name of the virus is sars19.

the name of the virus is sars19 and descrise name corona.

the name of the viurs is sars19and descrise name corona.

the name of the virus is : sars19

# Slicing

It is a concept of index of array(java)/list(python)

earth="it is the third planet of solar system"

print(earth[1]) 🡪 t

print(earth[-1]) 🡪 m

print(earth[:])🡪 it is the third planet of solar system

print(earth[4:10]) 🡪 s the

print(earth[4:-1]) 🡪s the third planet of solar syste

devops=("linux","bash script","jenkins","aws","anisable")

print(devops[0]) 🡪 linux

print(devops[1:3]) 🡪('bash script', 'jenkins')

print(devops[1:3][0]) 🡪bash script

print(devops[1:3][0][5:11]) 🡪 script

**we can slice it upto one character set {}, list [] , tripe (), dict {key:value}**

**note:**

**devops =(1,5,7,6,2,1,3,2)**

**devops1 =[1,5,7,6,2,1,3,2]**

**devops2 ={1,5,7,6,2,1,3,2}**

**print(devops)**

**print(devops1)**

**print(devops2)**

**output:**

(1, 5, 7, 6, 2, 1, 3, 2)

[1, 5, 7, 6, 2, 1, 3, 2]

{1, 2, 3, 5, 6, 7}

# Operators

Arithmetic Operators:

* Addition (+): Adds two values.
* Subtraction (-): Subtracts the right operand from the left operand.
* Multiplication (\*): Multiplies two values.
* Division (/): Divides the left operand by the right operand, yielding a floating-point result.
* Floor Division (//): Divides and returns the integer part of the result.
* Modulus (%): Returns the remainder after division.
* Exponentiation (\*\*): Raises the left operand to the power of the right operand.

a = 10

b = 3

print(a + b) # 13

print(a - b) # 7

print(a \* b) # 30

print(a / b) # 3.3333333333333335

print(a // b) # 3

print(a % b) # 1

print(a \*\* b) # 1000

Comparison Operators:

* Equal to (==): Checks if two values are equal.
* Not equal to (!=): Checks if two values are not equal.
* Greater than (>): Checks if the left operand is greater than the right operand.
* Less than (<): Checks if the left operand is less than the right operand.
* Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
* Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

x = 5

y = 7

print(x == y) # False

print(x != y) # True

print(x > y) # False

print(x < y) # True

print(x >= y) # False

print(x <= y) # True

Logical Operators:

* and: Logical AND, returns True if both operands are True.
* or: Logical OR, returns True if at least one operand is True.
* not: Logical NOT, returns the opposite of the operand's truth value.

p = True

q = False

print(p and q) # False

print(p or q) # True

print(not p) # False

Assignment Operators:

* =: Assigns a value to a variable.
* +=: Adds the right operand to the left operand and assigns the result to the left operand.
* -=: Subtracts the right operand from the left operand and assigns the result to the left operand.
* \*=: Multiplies the left operand by the right operand and assigns the result to the left operand.
* /=: Divides the left operand by the right operand and assigns the result to the left operand.
* //=: Performs floor division and assigns the result to the left operand.
* %=: Computes the modulus and assigns the result to the left operand.
* \*\*=: Raises the left operand to the power of the right operand and assigns the result to the left operand.

x = 10

x += 5 # x is now 15

x -= 3 # x is now 12

x \*= 2 # x is now 24

x /= 4 # x is now 6.0

x //= 2 # x is now 3.0

x %= 2 # x is now 1.0

x \*\*= 3 # x is now 1.0 (1.0 cubed)

Membership Operators:

* in: Checks if a value exists in a sequence (e.g., a string, list, or tuple).
* not in: Checks if a value does not exist in a sequence.

my\_list = [1, 2, 3, 4, 5]

print(3 in my\_list) # True

print(6 not in my\_list) # True

identity Operators:

* is: Checks if two objects have the same identity (i.e., they are the same object in memory).
* is not: Checks if two objects have different identities.

a = [1, 2, 3]

b = a

c = [1, 2, 3]

print(a is b) # True (a and b reference the same object)

print(a is c) # False (a and c are different objects)

print(a is not c) # True (a and c have different identities)

1. Bitwise Operators (for working with binary data, often used in low-level operations):
   * & : Bitwise AND
   * | : Bitwise OR
   * ^ : Bitwise XOR
   * ~ : Bitwise NOT
   * <<: Left shift
   * : Right shift

Example:

x = 5 # Binary: 101

y = 3 # Binary: 011

print(x & y) # Bitwise AND: 001 (decimal 1)

print(x | y) # Bitwise OR: 111 (decimal 7)

print(x ^ y) # Bitwise XOR: 110 (decimal 6)

print(~x) # Bitwise NOT: 11111010 (decimal -6 in two's complement)

print(x << 1) # Left shift by 1: 1010 (decimal 10)

print(x >> 1) # Right shift by 1: 10 (decimal 2)

These are the main categories of operators in Python. Understanding how to use these operators is fundamental to writing Python programs and performing various operations on data.

# Conditions

## If condition:

x=0

print("learning identation ")

if(x == 0):

print("if block is executing")

print("value is equeal to zero")

output

learning identation

if block is executing

value is equeal to zero

## else if condition

x=0

print("learning identation ")

if(x == 0):

print("if block is executing")

print("value is equeal to zero")

else:

print("else block is executing")

print("value is not equal to zero ")

print("out of if/else block")

output:

learning identation

if block is executing

value is equeal to zero

out of if/else block

## elif condition

x=40

if (x > 40):

print("x is greater then 40")

elif( x < 40):

print("x is less then 40")

else:

print("x is equal to 40")

output:

x is equal to 40

In [ ]:

a = input("enter the value of a :")

b = input("enter the value of b :")

c = input("enter the value of c :")

if(a>b and a>c):

print(a," is the greatest value")

elif(b>a and b>c):

print(b," is the greatest value")

else:

print(c," is the greatest value")

output:

enter the value of a :32

enter the value of b :20

enter the value of c :40

40 is the greatest value

"""

This script will implement our knowledge on

conditions and different datatypes.

"""

print("This IT Organization has various skill sets.")

print("Find out your match.")

print("Enter Capitalised Values: ")

DevOps = ["Jenkins", "Ansible", "Bash", "Python", "Puppet", "Dockers", "Kubernetes", "Terraform"]

Development = ("Nodejs", "Angularjs", "Java", ".net", "Python")

cntr\_emp1 = {"Name":"Santa", "Skill":"Blockchain", "Code":1024}

cntr\_emp2 = {"Name":"Rocky", "Skill":"AI", "Code":1218}

usr\_skill = input("Enter your desired skill: ")

#print(usr\_skill)

# Check in the database if we have this skill

if usr\_skill in DevOps:

print(f"We Have {usr\_skill} in DevOps Team.")

elif (usr\_skill in Development):

print(f"We have {usr\_skill} in Development Team.")

elif (usr\_skill in cntr\_emp1.values()) or (usr\_skill in cntr\_emp2.values()):

print(f"We have contract employees with {usr\_skill} skill.")

else:

print("Skill not found")

print("Please check if you have entered value in capitalize or check the spelling.")

# LOOPS

## For loop

planet = "earth"

for i in planet:

print(i)

print("print rest of the code")

output:

e

a

r

t

h

print rest of the code

## while loop:

x=0

while (x <= 10):

print("value of x: ",x)

print("looping")

x+=1

print("execute rest of the code")

output:

value of x: 0

looping

value of x: 1

looping

value of x: 2

looping

value of x: 3

looping

value of x: 4

looping

value of x: 5

looping

value of x: 6

looping

value of x: 7

looping

value of x: 8

looping

value of x: 9

looping

value of x: 10

looping

execute rest of the code

## nested for

"""

VACCINES = ["Moderna", "Pfizer", "Sputnik v", "Covaxin", "AstraZeneca"]

for vac in VACCINES:

print("")

print("I would like to take a shot of ")

for i in vac:

print(i)

"""

import time

x = 2

while True:

print("Value of X is:", x)

print("Looping")

x\*=2

time.sleep(1)

# break and continue

break condition

for i in "purandhar":

print(i)

if (i == "d"):

print("data is found")

break

print("out of the loop")

output:

p

u

r

a

n

d

data is found

out of the loop

continue statement

for i in "purandhar":

if (i == "d"):

print("data is found")

continue

print("the value of i : ", i)

print("out of the loop")

output:

the value of i : p

the value of i : u

the value of i : r

the value of i : a

the value of i : n

data is found

the value of i : h

the value of i : a

the value of i : r

out of the loop

import random

VACCINES = ["Moderna", "Pfizer", "Sputnik v", "Covaxin", "AstraZeneca", "CoronaVac"]

random.shuffle(VACCINES)

print(VACCINES)

LUCKY = random.choice(VACCINES)

print(LUCKY)

for vac in VACCINES:

print(f"\*\*\*\*\*\*TESTING VACCINE {vac}")

if vac == LUCKY:

print("###################################")

print(f"{LUCKY} Vaccine, Test SUCCESSFUL")

print("###################################")

print()

break

print("XXXXXXXXXXXX")

print("Test Failed")

print("XXXXXXXXXXXX")

print()

# build in function or methods

python as lot of build functions like:

and we not need byheart them

1)dir(option.) 🡪We will get the all the functions (inbuild for python3)

Ex: dir(random)

['BPF',

'LOG4',

'NV\_MAGICCONST',

'RECIP\_BPF',

'Random',

'SG\_MAGICCONST',

'SystemRandom',

'TWOPI',

'\_ONE',

'\_Sequence',

'\_Set',

'\_\_all\_\_',

'\_\_builtins\_\_',

'\_\_cached\_\_',

'\_\_doc\_\_',

'\_\_file\_\_',

'\_\_loader\_\_',

'\_\_name\_\_',

'\_\_package\_\_',

'\_\_spec\_\_',

'\_accumulate',

'\_acos',

'\_bisect',

'\_ceil',

'\_cos',

'\_e',

'\_exp',

'\_floor',

'\_index',

'\_inst',

'\_isfinite',

'\_log',

'\_os',

'\_pi',

'\_random',

'\_repeat',

'\_sha512',

'\_sin',

'\_sqrt',

'\_test',

'\_test\_generator',

'\_urandom',

'\_warn',

'betavariate',

'choice',

'choices',

'expovariate',

'gammavariate',

'gauss',

'getrandbits',

'getstate',

'lognormvariate',

'normalvariate',

'paretovariate',

'randbytes',

'randint',

'random',

'randrange',

'sample',

'seed',

'setstate',

'shuffle',

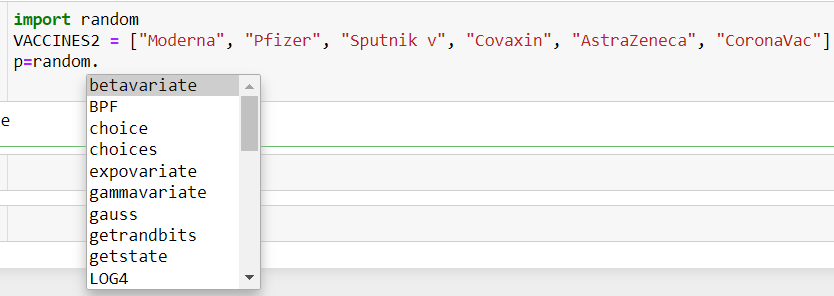
'triangular',

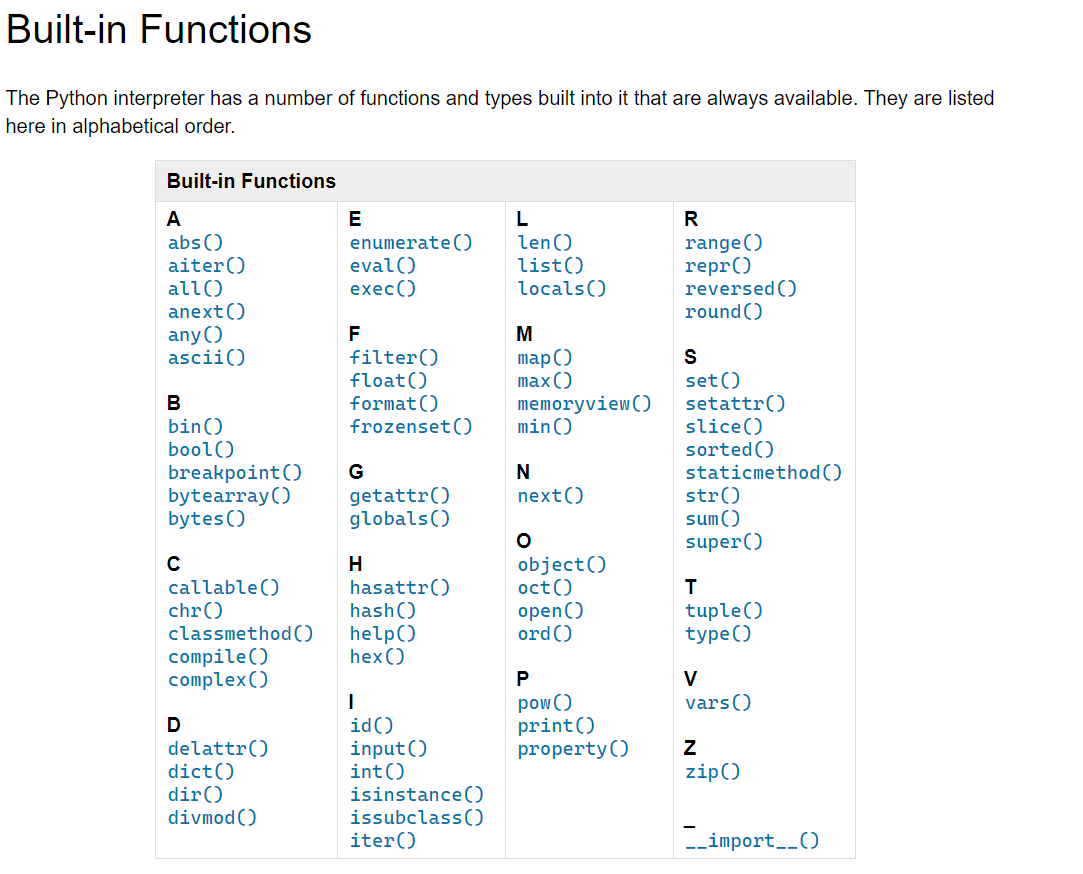
'uniform',

'vonmisesvariate',

'weibullvariate']

1. option.tab





name="purandhar"

print(name.capitalize())

print(name.islower())

print(name.count("0"))

print(name.replace("a","r"))

output

Purandhar

True

0

purrndhrr

learn 🡪 append ,extend , pop , insert(**insert(2,”geetha”**)) etc

list , dist

# functions

user define functions

def add(arg1,arg2) :

t=arg1+arg2

return t

add(2,5)

output : 7

def sum(arg):

x=0

for i in arg:

x+=i

return x

print(sum([4,5,6]))

output:

15

## Default argument

def Greeting(MSG = 'Morning' ):

print(f'good {MSG}')

print('welcome')

Greeting()

Output:

good Morning

welcome

# Keywords arguments

def vac\_feedback(vac, efficacy):

print(f"{vac} Vaccine is having {efficacy} % efficacy.")

if (efficacy > 50) and (efficacy <= 75):

print("Seems not so effective, Needs more trial.")

elif (efficacy > 75) and (efficacy < 90):

print("Can consider this vaccine.")

elif efficacy >= 90:

print("Sure, will take the shot.")

else:

print("Needs many more trials.")

#vac\_feedback("Pfizer", 95)

#vac\_feedback("Unknown", 45)

vac\_feedback(efficacy=34, vac="Unknown")

variable length arguments

**\***args(tripe) : we can take multiple value 🡪 non keyword argument

**\*\***kwargs(dict) : Variable Length Arguments \*\*kwargs 🡪(keyword Arguments)

Variable Length Arguments \*\*kwargs (keyword Arguments)

import random

def time\_activity(\*args, \*\*kwargs):

"""

Input: Multiple values for minutes, key=value pair activity

Output: Return sum of minutes + random minute spect on a random activity

"""

# print(args)

# print(kwargs)

min = sum(args) + random.randint(0, 60)

# print(min)

choice = random.choice(list(kwargs.keys()))

# print(choice)

print(f"You have to spend {min} Minutes for {kwargs[choice]}")

time\_activity(10, 20, 10, hobby="Dance", sport="Boxing", fun="Driving", work="DevOps")

# Modules

A modules will be python script or a package that has some functions or method available in that like random

🡪We can write our own modules

import modern 🡪it is pervious function that we had run

#print(dir(modern))

modern.order\_food("Salad", "Pizza", "Biryani", "Soup")

modern.vac\_feedback(efficacy=34, vac="Unknown")

modern.time\_activity(10, 20, 10, hobby="Dance", sport="Boxing", fun="Driving", work="DevOps")

from modern import \*

order\_food("Pizza")

vac\_feedback(efficacy=34, vac="Unknown")

time\_activity(10, 20, 10, hobby="Dance", sport="Boxing", fun="Driving", work="DevOps")

# OS TASKS

1. Add users
2. Add groups
3. Add users to group
4. Create directory
5. Assign user & group ownership to directory
6. Test if user or dir exists, if not create it
7. SSH in Python
8. Fabric
9. Webserver provisioning with python Fabric
10. Python Virtual env
11. Python for Various other tasks