**Python**

Python is a general-purpose, interpreted and object-oriented language. It’s weakly-typed and hence we don’t specify the type of variable before declaring any variable. All variables in Python are passed by reference.

**Data types:**

Following are the basic data types available in python

int, str, float, complex, bool (Can only be True or False)

Apart from the basic data types, python provides built-in objects: Numbers, Strings, Lists, Dictionaries, Tuples, File, Sets, Boolean, Modules, Classes, Functions

In terms of the core types, numbers, strings, and tuples are immutable; lists, dictionaries, and sets are not.

Statements:

A statement refers to a single line of execution in Python. We can have compound statement also, such as if/else block

Sample if-else statement

z = 0

if z > 0:

print("z is positive")

elif z < 0:

print("z is negative")

else:

print("z must be 0")

Notes:

* We can club multiple statements together using semi-colon as in: a = 1; b = 2; print(“ab”)
* A statement can be split in multiple lines by enclosing them in blocks which can be rendered by any type of brackets such as : if( A == 3 and

B == 5)

* A compound statement along with nested statement inside it can be coded in a single line . e.g. We can write – if x > 3: print(x). This allows us to write single line if/while/for loop statements
* An else: statement can be a part of try-except as well. An else: is executed if except: doesn’t catch an exception.
* Extended Unpacking: Sometimes in complex assignment statement, we need only the front and rest part rather than list whose count is same as that of len(<iterable\_obj>). In that case, we use extended unpacking statement in which variable preceded by \* is assigned rest of the string. For Example: (a, \*b) = (1,2,3,4) will assign (1) to (a) and (2,3,4) to (b).
* # += and extend allow any sequence, but + does not! e.g - <list> += “str” will work but <list> = <list> + “<str>” won’t
* Ternary expression:

if X: A = Y

else: A = Z

can be converted to: A = Y if X else Z

* A dictionary based switch statement:

print({'spam': 1.25,

'ham': 1.99,

'eggs': 0.99,

'bacon': 1.10}[choice])

**global and nonlocal statement**

The global statement and its nonlocal 3.X cousin are the only things that are remotely like declaration statements in Python. They are not type or size declarations, though; they are namespace declarations.

X = 88 # Global X

def func():

global X

X = 99 # Global X: outside def

Just like 'global' is used to refer to global variable, so is 'nonlocal' is applicable to variable declared inside the enclosing function. nonlocal is even more strict, though—scope search is restricted to only enclosing defs. That is, nonlocal names can appear only in enclosing defs, not in the module’s global scope or built-in scopes outside the defs

def tester(start):

state = start # Each call gets its own state

def nested(label):

nonlocal state # Remembers state in enclosing scope

print(label, state)

state += 1 # Allowed to change it if nonlocal

return nested

>>> F = tester(0)

>>> F('spam') # Increments state on each call

spam 0

>>> F('ham')

ham 1

> There is strict type-checking for 'nonlocal' variables because they need to exist in enclosing namespace but global needn't to be. Also, Python must resolve nonlocals at function creation time.

**Loops:**

We can iterate over a block of statements again and again using loops.

We can have 'else:' after 'while:' which is executed when while is exited without any break statement. Similarly, we can have 'else:' after 'for:' loop

for target in object: # Assign object items to target

statements # Repeated loop body: use target

else: # Optional else part

statements # If we didn't hit a 'break'

A sample for loop:

def countdown1():

for ct in range(9,0,-1):

print(ct,end=' ')

# or we can just pass an integer that will make index loop from 0 to that integer

for i in range(0, 4):

article = random.choice(articles)

A sample while loop:

def count\_down():

ct = 10

while ct>=1:

print(ct,end=' ')

ct = ct-1

**Scoping**

Three different scopes for variables:

> A variable name inside function is local to that function

> A variable is assign in enclosing def it is non-local to nested functions

> A variable assigned outside all defs is global to entire file. Here, global refers to module. A variable can be declared as global inside a function too using global statement.

Only actual name assignments determines the scope of variable.

Names are searched in the following order -> local -> enclosing functions -> global -> built-in

**Functions**

Functions are building blocks of program. There are majorly 2 type of functions: One that start with def: and the others that start with lambda:. The latter type is a type of anonymous function.

Sample function:

def name():

fname = input("Enter your first name: ")

lname = input("Enter your last name: ")

fullname = fname + " " + lname

print("Your name is: ", fullname)

THe def header line specifies a function name that is assigned the function object and that def header line is called a statement. Since this is a statement, it can be executed inside if/else condition too. 'def' assigns function object (series of statement constitutes them) to a name which can be assigned later to any other variable. A function can be assigned attributes too

def func(): ... # Create function objec

func() # Call object

func.attr = value # Attach attributes

We can directly call functions which are defined in another function.

**Arguments Passing**

In Python,

> Immutable arguments are effectively passed “by value.

> Mutable arguments are effectively passed “by pointer."

We can pass argument by keywords as: name=value

Arbitrary arguments in a function can be captured using an argument prepended b '\*'

The \*\* feature is similar, but it only works for keyword arguments—it collects them into a new dictionary, which can then be processed with normal dictionary tools.

>>> def f(\*\*args): print(args)

>>> f(){}

>>> f(a=1, b=2)

{'a': 1, 'b': 2}

Finally, note that keyword-only arguments must be specified after a single star, not two. Named arguments cannot appear after the \*\*args arbitrary keywords form, and a '\*\*' can’t appear by itself in the arguments list. Both attempts generate a syntax error:

>>> def kwonly(a, \*\*pargs, b, c):

SyntaxError: invalid syntax

>>> def kwonly(a, \*\*, b, c):

SyntaxError: invalid syntax

**Lambda Functions:**

lambda's general form is 'lambda' followed by one or more arguments, followed by an expression after a colon.

lambda argument1, argument2,... argumentN : expression using arguments

There is no explicit return statement. A sample can be to find square of an integer as:

l = lambda i: i\*i

**Iteration Protocol:**

Iteration constructs like for-in, generators, comprehensions calls iter() on iterable object and gets the iterator. After that they call \_\_next\_\_() method on iterator which is returned.

**Operators:**

The operators in Python are similar to what are in C/C++ except that it doesn’t contain increment operators. We have to use ‘//’ operator to carry out integer division

'==' operator performs equivalence check while 'is' operator check for identity whether 2 variable refers to same data.

>>> L1 = [1, ('a', 3)]

>>> L2 = [1, ('a', 3)]

>>> L1 == L2, L1 is L2 # Equivalent? Same object?

(True, False)

The case is different in case of strings because internally python caches strings

>>> S1 = 'spam'

>>> S2 = 'spam'

>>> S1 == S2, S1 is S2

(True, True)

**Functions**

**A packaged group of statements.**

**List**

List is a compound data type that can contain different type of data in a single variable. Lists are usually enclosed in square brackets as:

fam = [ “go”, 2, “po”, 2.34 ]

To check for an item in list:

**if “<Item>” in “<List>”**

To check for absence of item

**if “<item>” not in “<List>”**

Looping over list:

def count\_a(alist):

ct = 0

for let in alist:

if let == 'a':

ct = ct + 1

print("There are",ct,"letter a's in the list.")

**List Slicing**

We can carve out a part of list using range specifiers:

name[start : last] #start and end are indices in list, with start included and end isn’t

Skipping start index will start including from first element as in name[:5]. While skipping end index will include everything starting from the given vertex name[2:]

**Assignment**

Let’s say we have a list x and we do y = x; then ‘y’ will contain just the reference of list and not the copy of list using equal sign to assign one list to another will just copy the reference and not the complete list. To realize that, you can assign list as:

y = x[:]

y = list(x)

List Comprehension:

It is a way to create a new list using existing list. It begins with an arbitrary expression which contains the loop variable.

[x+10 for x in <iterable\_object>]

Filter clause in list comprehension (If: )

As one particularly useful extension, the for loop nested in a comprehension expression can have an associated if clause to filter out of the result items for which the test is not

lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']

Nested loops in comprehension:

[x + y for x in 'abc' for y in 'lmn']

['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

**Deletion in list**

You can also remove elements from list using del() function as in:

**x = ["a", "b", "c", "d"]**

**del(x[1])**

Notes:

* A list can contain more lists.
* We can remove a section from a list (between indices I and j) as: L[i:j]=[]
* We can use negative indices to refer to the elements inside the list, -1 will correspond to last element, -2 to the second-last elemenet and so on.
* In Python, every tool that scans an object from left to right uses the iteration protocol.
* To pre-allocate some space in list, you can do: l = 100\*[None]
* Slicing out of range indices will give entire list e.g. l = [1,2] ; l[-9:78] = [1,2]
* List append() function doesn’t iterate while extend() function do.

**Methods:**

**list.index(<element>)** # finds/searches index of <element> in list, if item is not there in list, ValueError is thrown

**list.count(<element>)** # returns count of <element> in list

**list.append(<element>)**

# To append items of other list to current one, use extend method

**list.extend(<another\_list>)**

**list.remove(<element>)** # Raises ValueError in case <element> doesn’t exist

**list.reverse()**

**list2 = list.copy()** # Creates copy of list and returns the same**.**

# To remove an element from list

**list.pop(<index\_to\_remove>) or del list[<index\_to\_remove>]**

**list.sort(key=, reverse=) # Sorts the list, key = str.lower makes case insensitive sorting (It’s called keyword arguments)**

**We can also pass reverse=True to reverse sort the list.**

**sorted(<data\_structure>) can sort a list which might have been created via comprehensions too which makes it more versatile than list.sort() method.**

**Strings**

We can use both single quote and double quote to represent a string. Strings are immutable. Hence, assignment, such as: s[0] = “w” will fail. Strings also support raw strings wherein special meaning of '\n', '\t' etc. are ignore. It's specially useful in directory paths such as C:\node\temp.

Methods available with strings:

**str.capitalize()**

**str.replace(“s”, “sa”)**

**str\_up = <string>.upper();**

**str.lower()**

**str.find(<substring>)**

**str.replace(<old\_str>, <new\_str>)**

**str.split(<delimiter>)**

**str.upper()**

**str.lower()**

**str.isalpha/isdigit/isspace/islower/isupper/isdigit/isdecimal/istitle/isalnum/isnumeric()** # returns a boolean value

**str.rstrip()** # removes whitespace characters from right

'{m} {m+1} … '.format(str1, str2...)

**str.encode('utf8'/'utf16')**

**Notes:**

Triple slicing:

* S[i:j:k] slices the string between range i to j (not including j) separated by k steps

**{0:<25} # Left-align with 25 spaces**

**{0:>25} # Right-align with 25 spaces**

**{0: 25} # Left-align(String) or Right-align (An integer) with 25 spaces**

**Tuple**

It is like a list only and it’s just that it is immutable. Also, 'tuple' object has no attribute 'append'. A tuple is typically contained in a pair of parentheses. One element tuple is represented as (<item>,) because (<item>) will spit out that item only. You can omit parentheses when assignment is not ambiguous as:

S = 1,2,3,4

**tuple.count(<element>) # Count number of entry in tuple**

**tuple.index(<element>) # Tells the index of element in tuple**

**Notes:**

**> To sort a tuple, you can either first convert it into a list and call sort() method or call sorted() method which works on sequence data structures.**

**Dictionary:**

It is an associative container just like an unordered\_map in C++. Each dictionary entry is just a tuple. Its type name is ‘dict’.

Dictionaries can be appended to. Items cannot be retrieved by item number, because the items have no inherent order. The values can be retrieved by using their keys. Dictionaries have no notion of order and hence cannot be sliced ([3:7] doesn't mean anything) and they cannot be strided -- it might appear that you can, but the order is unpredictable.

To loop over dictionary:

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

**print(key, "--> ", value)**

**for item in d.items():**

**print(item) # Each item is a tuple & can be referenced by offset**

# To retrieve the keys:

**dict.keys()** # It returns an iterable object

# To retrieve the items:

**dict.items()**

# To retrieve the values:

**dict.values()**

# To delete a key

**dict.pop(key) or del dict[key]**

# To delete all elements

**dict.clear()**

# To create dictionary from key\_list and value\_list

**d = dict(zip(keyslist, valueslist))**

# To create dictionary in one go:

**d = dict( name='bob', job='dev', age=40)**

# zipping key and value list

**d = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40]))**

# To create a dictionary with a given value and multiple keys

**dict.fromkeys(['a', 'b'], 0) 🡺 {'a': 0, 'b': 0}**

**Notes:**

> Only immutable objects can be used as keys. Mutable objects such as lists, sets, and other dictionaries don’t work as keys,

but are allowed as values.

**Sets:**

Set is an unordered and immutable data structure. You can make a set out of a sequence (such as string) as

**X = set(‘spam’)**

# Make a set with set literals

**Y = { ‘a’, ‘b’, ‘c’ }**

We can carry out the following operations with sets:

**Intersection (X&Y), Union (X|Y), Difference (X-Y), Superset (X>Y)**

# Finding out differences

**set('spam') - set('ham') 🡺 {‘p’, ‘s’ }**

# Filtering out duplicates

**list(set([1, 2, 1, 3, 1])) 🡺 { 1, 2, 3}**

**Built-in functions:**

# To know the type of variable

type(<variable\_name>);

# To convert types of variable into another

str() , int() , float() etc

# To return the list of integer starting from start\_index up to, but not including, end\_index with a given step (increment)

range(start\_index, end\_index, step); -> returns list

#To round up the given number

round(<float>, <precision>);

# To return the max/min/sum of the list

max/min/sum(list);

# To round to closest integer

round(<number>);

# To get info regarding a function

help(<function\_name>);

# To get the memory address of object

id(<object>)

# To sort a list

sorted(<list>)

# To reverse a list

reversed(<list>)

# To list down all the attributes in a filename/of an object

dir(<module\_name>)

# To retrieve the integer-ordinal value of character

ord(<char>)

# To return binary representation of an integer

bin(integer)

# To convert source to equivalent Python built-in data type

eval(<source>)

# To check if all objects in list give bool(object) as true

all(<iterable>)

# To check if any object in list gives bool(object) as true

any(<iterable>)

# zip function takes n lists and joins them into 1 with each item of new list being a tuple containing one item from each list. Moreover, zip truncates result tuples at the length of the shortest sequence when the argument lengths differ.

zip(<list1>, <list2>, <list3>,...)

# Map performs the given <operation> on each element of <list> and stores the result in a new list. It’s a built-in that applies a function call to each item in the passed-in iterable object. map is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also returns an iterable object itself in Python 3.X, so we must wrap it in a list call to force it to give us all its values at once.

list(map(<operation>, <list>))

# Enumerate function takes a list and returns a generator object which returns result in the form (index, value).

enumerate(<list>)

For example:

S = 'spam'

for (offset, item) in enumerate(S):

print(item, 'appears at offset', offset)

# To return the next item returned by the enumerator which is working on list. each result in the form (Index, Item).

next(<enumerate>) or next(<file\_object>)

When file object is passed to next function, it invokes readline() function

# About print() function

print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])

Values after '=' are default arguments. Also, we can redirect the outout of print statment to another stream by resetting th sys.stdout object.

To run 3.X's print() function in 2.X, use : from \_\_future\_\_ import print\_function

# To get iterator whose \_\_next\_\_() method we can call later on

iter(<Iterable\_object>) # We don’t need to call this method on files because they are themselves iterators

reduce(<function>, [list])

# reduce function takes a function and a list and returns a single result. It is available in the functools module in 3.X, Here are two reduce calls that compute the sum and product of the items in a list. E.g. –

>>> from functools import reduce

>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])

10

>>> reduce((lambda x, y: x \* y), [1, 2, 3, 4])

24

# returns items in an iterable for which a passed-in function returns True

filter(<fn\_name>, <list>)

List Comprehension:

It applies and expression to a list. The operations performed by map() and filter() can be concisely implemented by LC

[expression for target in iterable if condition]

We need to have multiple loops in LC to use it over dictionary

res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]

[100, 200, 300, 101, 201, 301, 102, 202, 302]

**Generator functions**

These are the functions which can be suspended during execution and made to return intermediate result and resume from the place from where it left off e.g.

def gensquares(N):

for i in range(N):

yield i \*\* 2 # Resume here later

>>> for i in gensquares(5): # Resume the function

print(i, end=' : ') # Print last yielded value

**Generator Expression:**

These are almost similar to List comprehension. It's just that it returns a generator object which then behaves just like a generator functions in terms of functionality.

>>> (x \*\* 2 for x in range(4)) # Generator expression: make an iterable

<generator object <genexpr> at 0x00000000029A8288>

Notes:

Calling some built-in functions such as append, extend, sort etc. modify the list in-place, but these functions returns None. Hence, call in-place change operations without assigning their results.

> In 3.X zip, map, and filter do not support multiple active iterators on the same result.

**Packages**

Packages are the backbone of Python because not everything is available as a core language feature

**import math** will import the complete math package

**from math import pi** will just import pi function from math package

**Random:**

***import random***

**random.seed()** # Seeds the random number generator

**random.random()** # Generates random number between 0 and 1

**random.choice(<list>)** # returns one of the element in list

**random.randint(<start>, <end>)** # Generates random number between start and end

**string str = random.choice(<list>)** # Returns the string from a list

**statistics**

**statistics.mean()**

**statistics.median()**

**statistics.mode()** # Most often repeated element

**statistics.stddev()**

**statistics.variance()**

**Math**

**math.pi # An attribute**

**math.sqrt(<number>)**

**import Fraction**

from fractions import Fraction

f = Fraction(2, 3)

f+1 🡺 Fraction(5, 3)

f + Fraction(1, 2) 🡺 Fraction(7, 6)

**import pickle**

The pickle module is a more advanced tool that allows us to store almost any Python

object in a file directly, with no to- or from-string conversion requirement on our part.

It’s like a super-general data formatting and parsing utility. To store a dictionary in a

file, for instance, we pickle it directly:

F = open('datafile.pkl', 'rb')

E = pickle.load(F)

E will contain {'a': 1, 'b': 2}

**import json**

Python dictionary is almost equivalent to JSON.

>>> import json

>>> rec{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}

'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'

>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)

>>> P = json.load(open('testjson.txt'))

>>> P

{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}

**import collections**

Collections is a part of standard python library. We can use namedtuple that adds logic to tuples so that they can be accessed by both index and keys.

>>> **from collections import namedtuple**>>> **Rec = namedtuple('Rec', ['name', 'age', 'jobs'])**>>> **bob = Rec('Bob', age=40.5, jobs=['dev', 'mgr'])**>>> **bob**Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])

>>> **bob[0], bob[2]**('Bob', ['dev', 'mgr'])  
>>> **bob.name, bob.jobs**('Bob', ['dev', 'mgr'])

**import struct**

struct module knows how to both compose and parse packed binary data. In a sense,

this is another data-conversion tool that interprets strings in files as binary data.

>>> F = open('data.bin', 'wb') # Open binary output file

>>> import struct

>>> data = struct.pack('>i4sh', 7, b'spam', 8) # Make packed binary data

>>> data

b'\x00\x00\x00\x07spam\x00\x08'

>>> F.write(data) # Write byte string

To read data:

>>> F = open('data.bin', 'rb')

>>> data = F.read() # Get packed binary data

>>> data

b'\x00\x00\x00\x07spam\x00\x08'

>>> values = struct.unpack('>i4sh', data) # Convert to Python objects , '>' means Big-Endian, '<' refers to little

# endian '@' and '=' refers to native order

>>> values

(7, b'spam', 8)

**import copy**

Normal copy operations won't work in case of nested data structures. So, use deep copy instead of normal copy.

X = copy.deepcopy(Y)

**NumPy**

***import numpy***

NumPy advantage is that you can do efficient and good operations over the entire list in one go using their package such as finding out BMI(Body Mass Index) using arrays of weight and height.

Numpy arrays cannot contain elements of different types.

We can create 2-D NumPy arrays too. Each element of a 2-D NumPy array can be accessed as arr[i][j] or arr[i, j]

<array\_name>. Shape returns the dimensions of 2-D array

**Exception Handling:**

**try:**

**<code>**

**except <error> as e:**

**<code>**

**Working with files**

Python provides concise and utility functions to do file handling. Some of them are:

**file\_handle = open(<Filename>[, ‘w’)**

**file\_handle.close()**

You can iterate over the file using a simple loop as:

**for row in file\_handle**

**for line in open('myfile.txt'): # Use file iterators, not reads**

**print(line, end='')**

# You can specify the files while executing the python script. To access that functionality you need to:

**import sys**

Then , sys.argv[1] will be the first argument passed along with the file.

Preopened file objects in the sys module, such as sys.stdout

**Working with CSV files**

To work with CSV files, you need to:

**import csv**

csv.reader(<file\_handle>) return rows of list

**f = open(filename)**

**for row in csv.reader(f):**

**print(row[0], row[1], row[2])**

**f.close()**

To write CSV data to csv file

**f = open(filename, 'w', newline='')**

**for item in L:**

**csv.writer(f).writerow(item)**

**f.close()**

**Comprehension Expressions:**

A powerful technique to extract data from matrix. These are of the form:

[ <expression> for <element> in <list> if <conditional\_expression> ]

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

[row[1] + 1 for row in M] will result in [3, 6, 9]

**Object Oriented Programming in Python**

class C2: ... # Make class objects (ovals)

class C3: ...

class C1(C2, C3): ... # Linked to superclasses (in this order)

I1 = C1() # Make instance objects (rectangles)

I2 = C1() # Linked to their classes

**Notes:**

Use ‘#’ as leading character to add comments in python code

Add end=’ ’ o make print not add new line.

We can write import statement inside the function also

String is left aligned and number is right aligned by default

[] is an empty list; () is an empty tuple; and {} is an empty dictionary.

> Only mutable objects—lists, dictionaries, and sets—may be changed in place; you cannot change numbers, strings, or tuples in place. 'bytearray' string type is mutable.

>>>

> The variable'\_' has special meaning and printing it prints the last result

\* Lists, dictionaries, and tuples can hold any kind of object.

\* Sets can contain any type of immutable object.

\* Lists, dictionaries, and tuples can be arbitrarily nested.

\* Lists, dictionaries, and sets can dynamically grow and shrink

> Syntactically, nested objects are internally represented as pointers to separate pieces of memory.

> By default, assignment creates a reference to already existing object. To create a new copy, either call copy() method of

data type or call in-built functions as list(<orig\_list>), dict(<orig\_dict>),set<orig\_set>

> A = L[:] will create a copy of L and assign that to A

> Comparison between mixed/unrelated types is forbidden in Python

> Comparison between dictionaries is not allowed in Python 3.X

> In interactive mode, the last printed expression is assigned to the variable \_

> If you don’t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

> To delete a redefinition of a variable

del <var\_name>

from <module> import <name1>, <name2>... does the following:

: import module

: name1 = module.name1

: name2 = module.name2

: del module # get rid of module

> To use reload function:

from imp import reload

reload(<module\_already\_imported>)