# ­­­­­Introduction:

Python is Interpreted, Interactive, OO (object oriented).

Python features:

1. Simple
2. Object oriented
3. Vast library of add-on modules
4. Open source – supported by large groups, continuous development.
5. Platform independent­­­
6. Dynamic – Runtime flexibility
7. Very-high-level programming – can build high level solutions from other programs.
8. It offers basic text manipulation like Perl, awk and other OS services like advanced programming.

**Simple – there is no need to explicitly manage memory and pointers, It relies on a few core data structures and statements. Python conceals the mechanics of object references from the programmer, making it impossible to corrupt a pointer.**

Python uses an English-like syntax, there are few syntax rules, and indentation makes the structure of the software very clear.

Object-Oriented – **Everything in python is an object**. This is distinct from Java and C++ which create confusion by having objects as well as primitive data types that are not objects. Everything is derived from the base class type. The class int is also derived from the base class type.

Security in python – Python is obscure, it hides something but it doesn’t mean you cannot access that hidden thing, you can access it in a different way.

Hardware --> Machine Language --> Assembly Language --> High-level language.

Assembly language is low level as it is closer to machine language. High level language is closer to human language.

Compilers convert source code --> object code.   
Compiler differs from an interpreter, which analyses and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately in one step without compiling. Compilers require some time before an executable program emerges. The compiled binary code is platform dependent. Interpreted can be platform independent.

Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

Source code --> Preprocessing --> Machine Language

Source code --> Processing --> Interpreter

Python is a higher level language than C/C++ because you don't have to interface with the hardware directly (like managing memory).

Python is byte code interpreted. Source code is not compiled to machine language, it is compiled to intermediate bytecode. This code is then executed by Python's virtual machine.

Whether a particular language implementation works by compilation or interpretation is merely an implementation decision, because any language can be implemented either way.

Python enables programs to be written compactly, hence programs written in Python are typically much shorter than equivalent C, C++, or Java programs, the reasons are:

1. The high-level data types allow you to express complex operations in a single statement
2. Statement grouping is done by indentation instead of beginning and ending brackets
3. No variable or argument declarations are necessary(dynamic typing).

Difference between syntax and semantic:

Syntax is about the structure or the grammar of the language. It answers the question: how do I construct a valid sentence. Semantics is about the meaning of the sentence. It answers the questions: is this sentence valid? If so, what does the sentence mean?

Lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens. lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis.

**How does indentation work in python:**Changes to the indentation level are inserted as tokens into the token stream. The exact amount of indentation doesn't matter at all, but only the relative indentation of nested blocks (relative to each other).

The lexical analyzer (tokenizer) uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed on the stack, and an "INDENT" token is inserted into the token stream which is passed to the parser. There can never be more than one "INDENT" token in a row. When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a "DEDENT" token is generated. Obviously, there can be multiple "DEDENT" tokens in a row. At the end of the source code, "DEDENT" tokens are generated for each indentation level left on the stack, until just the 0 is left.

>>> if foo:

... if bar:

... x = 42

... else:

... print foo

...

<if> <foo> <:> [0]

<INDENT> <if> <bar> <:> [0, 4]

<INDENT> <x> <=> <42> [0, 4, 8]

<DEDENT> <DEDENT> <else> <:> [0]

<INDENT> <print> <foo> [0, 2]

<DEDENT> [0]

NEWLINE, INDENT and DEDENT, identifiers, keywords, literals, operators, and delimiters are the tokens in python. Whitespace characters are not tokens, but serve to delimit tokens.

Is using tab as indentation a bad thing – If you type code in the interpreter, it is good. But tabs may be interpreted differently by different editors. So using 4 spaces as indentation is good.

**Indentation Advantages and disadvantages:**

To tell when a block ends, we generally use curly braces. To improve readability along with curly braces, we also use indentation. This makes it redundant. If you can only use indentation – redundancy wouldn’t be there.

# Python Identifiers and keywords:

* Identifies a variable, function, class, module, or other object.
* No punctuation characters such as @, $, and % within identifiers.
* Case sensitive
* Class names - MyClass.
* All other identifiers start with a lowercase letter.
* \_private - internal variable.
* \_\_stronglyPrivate identifier.
* Ends with two trailing underscores, the identifier is a language-defined special name. languageName\_\_
* Python will print newline auto, no need to specify \n, unless it is not in the same print statement. If you want a space, after print add “,”

Print “hello”,

* Inside single quotes, if double quotes are used, they will be printed as is. Same for single quotes used inside double quotes. But if inside double quotes you use double quotes, those double quoted won’t be printed.
* Inside triple quotes, you need not escape anything.
* Standard indentation requires Python code to use four spaces.
* Don’t use semicolon in python – Never. Only indentation is to be used.
* No Curls, no braces. Only line indentations
* \_ is limited to interpreter only. It stores the last evaluated value.
* \_name = name is to be treated as “private” by a programmer, so that the next person (or yourself) using your code knows that a name starting with \_ is for internal use.
* \_\_\*\_\_ = special method names.

Python statements typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue without going to newline. whereas for triple quotes, it prints in the newline , it takes the string as it is.

Triple quotes for paragraph ( “”” “””). Double for string. Single for word. Single can also be used for string, but double is more readable.

String has to be included in quotes, if not, it is an error.   
A group of individual statements, which make a single code block are called suites in Python.

nonlocal in python implies that if you make variable nonlocal you can write to it(make changes to it) from the nested function. By default read is available for the nested function i.e. the nested function can read variables defined in the outer loop.  
Keywords:

False class finally is return

None continue for lambda try

True def from nonlocal while

and del global not with

as elif if or yield

assert else import pass

break except in raise

# Python Modules:

* Sys – modifying variables and functions of the interpreter.
* Getopt - parse command-line options and arguments.
* math
* Os – os dependent functionality
* Csv
* Re
* textwrap

sys.argv is the list of command-line arguments.len(sys.argv) is the number of command-line arguments. sys.argv[0] is the script name.

sys.exc\_info() gives a tuple of three values that give information about the exception that is currently being handled.   
sys.getsizeof(variable\_name) gives the size of the variable in bytes.

sys.getrecursionlimit() #gets the recursion limit to avoid crashes  
sys.setrecursionlimit(limit\_value)

os.chdir/mkdir(dir\_name). os.path.exists(path\_name). os.path.isfile(filename\_with\_path). os.remove(filename). os.rmdir(dir\_name). os.getpid(). os.system(command\_name)  
All the inputs to os. commands should be strings.

os.path.basename(path\_name)

os.path.dirname(path\_name)

Gives the base directory name and the directory name respectively.

os.walk(path\_name) generates a tuple of length 3, first value – path\_name, second value – list of folders present in the path, third value – list of files present in the path.

Os.system is replaced by subprocess.call()

# Variables

Variables are nothing but reserved memory locations to store values. Type of variable tells us how much memory is to be allocated to that variable. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. This concept is similar to that of allocating lockers, small medium large etc. Python is completely object oriented, and not "statically typed", it is dynamically typed. So the declaration happens automatically when you assign a value to a variable.

In python, The variables are Implicitly defined, Strongly typed.

2 things:

1. Weakly typed vs strongly typed
2. Statically typed vs dynamically typed

In a weakly typed language a compiler / interpreter will sometimes change the type of a variable. For example, you can add strings to numbers 'x' + 3 becomes 'x3'. Instead of raising an exception, execution will continue but your variables now have wrong and unexpected values. the compiler or interpreter can perform behind-the-scenes conversions to make these types of operations work; You cannot do anything that's incompatible with the type of data you're working with.

In a strongly typed language you can't perform operations inappropriate to the type of the object – this is easier, because the exception is raised. variables will not be silently changed without your knowledge.

Weak types automatically try to convert from one type to another, depending on context (e.g. Perl). Strong types never convert implicitly.

In a statically typed language, the type of variables must be known (and usually declared) at the point at which it is used. You cannot use it without declaring it.

In a dynamically typed language- a variable is simply a value bound to a name, names have no types but objects still have a type, but the type is determined at runtime. You are free to bind names (variables) to different objects with a different type. Only perform operations valid for the type. A statically typed language can catch some errors in your program before it even runs, by analysing the types of your variables and the ways they're being used. A dynamically typed language can't necessarily do this, but it raises an exception.

Objects are Python’s abstraction for data. All data in a Python program is represented by objects.

Every object has an identity, a type and a value.

Identity – memory address.   
Type – datatype of the object.

Once an object is created, it’s identity and type can’t be changed, value may or may not be changed.  
if value is changed, it is mutable, else immutable.

The value of an immutable container object that contains a reference to a mutable object can change when the latter’s value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.

Assigning a variable does not mean that variable is the object. We used very specific language noting that assignment statements bind a name (identifier) to an object. Variables can be reassigned.

objects containing references to other objects are called containers. Eg-tuples, lists and dictionaries.

Every object is stored in a memory location. By giving a variable name, we give a name to reference to that memory location. Containers are a collection of such objects.

# 5 data types in python

1. Numbers
2. String
3. List
4. Tuple
5. Dictionary

Numbers have subtypes– int, float, complex, long int

Conversion between data types – datatype\_name(variable\_tobe\_converted)

If you convert integers to string, and then add them, they will not give integer sum, instead they give concatenated string type. To avoid this while doing sum, do int(x) + int (y)

None is a data type in python, it is NoneType

Duck typing:

"If I access a property on this object and it returns something back to me, then I can assume it has this property."

the fact that you know what something is by describing how it acts. This is the foundation of the so-called "duck typing"

The idea is that it doesn't actually matter what type my data is - just whether or not I can do what I want with it. The expression 3 + 3 is syntactic sugar for calling the \_\_add\_\_ method of the integer type. It's the same as calling int.\_\_add\_\_(3, 3)

EAFP is a Python acronym that stands for easier to ask for forgiveness than permission.

Metaprogramming:

Meta is referring to itself.  
Eg - "programming" is applying transformations to data. Just like that, you may also run some code to change the code itself.

Class is callable, but the object of the class is not callable, to make it callable use \_\_call\_\_method.

the built-in function type() returns the content of the magic attribute \_\_class\_\_ (magic here means that its value is managed by Python itself offstage).

You can create a blank list by doing blank\_list=list()

This is because list is a class and you are creating an empty list by calling the class “list”-->list()

# Int

divmod(self, value) #returns tuple(quotient,remainder)

# **String**

It is enclosed in single quotes or double quotes.

str[0] # Prints first character of the string

str[2:5] # Prints characters starting from 3rd to 5th

str[2:] # Prints string starting from 3rd character i.e.2nd position

str \* 2 # Prints string two times

str + "TEST" # Prints concatenated string

len(string\_name) gives number of characters in a string.

**String methods:**

If s is string name

1. s.index(“substring”)   
   #gives the first index of substring in the string.
2. s.find(str,begin\_position,end\_position)

#from the begin to end-1, search is made and if the string is found in that range, index is returned, else -1 is returned. #Difference between find and index is that, index gives ValueError if substring is not found, whereas find just returns -1. If you are sure that the substring is present in the string, use index.

1. s.replace(‘old’,’new’,’max\_count to be replaced’)  
   #this will return a new string, the original string is the same.
2. s.format

#“sum of 1,2 is {0}, difference of 1 and 2 is {1}”.format((1+2),(1-2)). This is also considered as string method.

1. s.format\_map

#is similar to s.format(m) except that m is used directly and not copied to a dict.

1. s.swapcase()
2. s.count("substring",begin\_position,end\_position)   
   #Will count the number of hits of substring
3. s.upper()   
   #Prints uppercase substitute of the string
4. s.lower()
5. s.startswith("Hello")   
   #If it starts , then returns TRUE, else False
6. s.strip()  
   #strips out the white spaces from left and right, not in the middle, only left and right (lstrip, rstrip respectively)

print "%03d" % 5

005 ===will make a number 3 digit, will pad with 0 if it is not 3 digit

String splicing – if you want 0th to 5th indexes [0:6]. Logically it is [0:6)

Use escape character if you want to escape single quotes or double quotes.

You cannot access string in reverse direction. Only indices are negative.  
s[-1:-3] is wrong.

If str[10] is given, if 10 index is not there, it will give traceback error. But if you slice with 10, no error. Str[10:] – you get an empty string ‘’. But no error. Because you are trying to access the slice of a list at a starting index which is greater than the number of members in the list. Such slice doesn’t exist, so empty list.

“extended slicing”: a[i:j:k]   
selects all items of a with index x where x = i + n\*k, n >= 0 and i <= x < j. Select i to j indices, in steps of k.

Strings are immutable, it means that particular index of a string cannot be changed, you can change the whole string. Same concept as deep copy and shallow copy.

Shallow copy –same hash – like aliasing

Deep copy – new object.

To modify a string – you can convert to a list and then use join

Input of join --> list output --> string

Input of split --> string output --> list

Join is string function – which is operated on a list.

Input of join is list, but that list should contain only string items. If it contains int, it cannot be joined.

Input of join can also be a string.

Else slice the string on either sides of the character to be modified and concatenate those three strings

Just like split generates list after splitting, partition generates tuple after partitioning. Syntax t=s.partition(‘,’)

Partition separates the string in two and returns a tuple of 3 elements, the first portion of partition, the string based for the partition, the second portion of the partition.

bin(n),oct(n),hex(n) will give bin,hex,oct values of n= built\_in functions

Difference between S.title() and S.capitalize() is that, title will convert ‘3g’ to ‘3G’. Capitalize will convert ‘3g’ to ‘3g’. i.e. Title omits numeric values, capitalize considers numeric values as well. If space is the first character, capitalize will not change the case. Title will change the case for the first char for all the delimited words.

Don’t play around with builtins. If you define str and assign a value to it, the ‘str’ function couldn’t be called because str now becomes an object.

Print is a function in python3, in python 2 it is a statement. So include everything in braces.

print("%d" %(i)).

But this is not a good way, so use format.

print("number is {}".format(i)).

For two variables, print("number is {} {}".format(i,j))

Inside the print function call, format resides. But not inside “”.

When dealing with int, be very careful to convert it to string type if you are going for concatenation. ASCII codes -8-bit, meaning that bytes could hold values ranging from 0 to 255(2^8=256). Unicode started out using 16-bit characters instead of 8-bit characters. 16 bits means you have 2^16 = 65,536

Format and argument specifiers are properties of the string, so they can be used anywhere where the strings are used.

print u'Hello, world!'

To Swapcase without using the builtin swapcase method , this helps, because the int value for lowercase letter-int value for uppercase letter = 32.

ord(‘A’) = 65 ord(‘Z’) = 90  
ord(‘a’) = 97 ord(‘z’) = 122

ord(‘z’)-ord(‘Z’)=32

# List

A list contains items separated by commas and enclosed within square brackets []. To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. Slice operator is similar to that of strings.

# If you declare a list, it’s values should be enclosed in [], when you use the name of the list further, that name, need not be enclosed in [],as we do in perl – every time we call an array name , you write it as @arrayname

mylist.append(variable\_to\_be\_appended)

list\_name.insert(index,obj) #this will insert obj at index position index.

Looping through the list using index position:  
for i in range(len(a)):  
print a[i]

a is the list name

**cmp(list1,list2)**

**max/min(list1)**

**del list1**

List1.**append(obj)**

List1.**count(obj)**

List1.**extend(List2)**

List1.**index(obj)**

List1.**insert(index, obj)**

list1.**reverse()** = if you do print list1, you get the reverse list

list1.**remove(obj\_name)**

list1.**pop(index\_id)**

list1.**sort()** – sorts the list in increasing order

list1.copy() – shallow copy

By default list sorts in ascending order, if you do sort(reverse=True) then it sorts in the descending order.

Sorted is a builtin fn, whereas list.sort() is a method of list. list.sort() method is only defined for lists, sorted() function accepts any iterable. Both accept the arguments, key= and reverse=

List.sort() sorts the list in place, whereas sorted creates a new list and sorts it.

sorted(iterable, key=None, reverse=False)

L.sort(key=None, reverse=False)

Sorted works on any iterable, but returns only list.

key parameter should be a function that takes a single argument and returns a key to use for sorting purposes.

Eg- key=lambda student: student.age

(Or) sorted(student\_list, key=lambda student: student[2])

To sort the nested list, based on student[2] i.e. 3rd item of each nested list.

This is similar to

Student\_list.sort(key=itemgetter(2))

itemgetter is in module operator.

Difference between pop and remove is that, for pop you need to give index id, for remove you need to give object name. Default item popped using pop is the last item of the list.

To append more than one item to a list, do list += v1,v2 etc

Append and extend both are same, only difference is that, when you append a list to our list, append adds [] too, whereas extend just adds the elements. list += v1,v2 is example of extend. Both accept any iterable as input.

list\*2 will print the list two times, it wont multiply twice, it is not item by item multiplication.

Len(object) = len is built in function, it is applicable to all the datatypes like string, list, tuple etc..

List can have duplicates. The duplicate is treated as different as it has diff index.

list is similar to stacks(LIFO)

Matrix using list:

If m is the matrix , len(m) gives the number of rows.  
Len(m[0]) gives the number of columns in the first row

In Python, variables are linked to objects by references for the case of lists. If x=y, if you change y, the same would be affected on x.

list(enumerate(list\_name)):

Will start indexing the values in list\_name from 0. 0 is default

list(enumerate(list\_name, start=1)) - Will start indexing the values in list\_name from 1.

We are using list() becasue enumerate gives an emurate object and you have to convert it to list.

list2 = list1 #Here both the lists are sharing the same address, so if you make any changes in list2, the same changes will be affected in list1 and vice-versa as the address is same.

list2 = list(list1) #Here the address is different. The copy of list1 is assigned to list2. Hence the address of list1 remains the same, whereas the address of list2 is different.

If we do left=right, the value of right is existing and this is assigned to a new variable in the left. If the variable in the left is already created, it is overwritten in the case of lists.

Memory is allocated to values, not the variable names.

If we do x=5, y=5, the id’s of both x and y would be same.

**List comprehension**: to write multiple lines of code to a single line

How does list comprehension work. It is in lexographical order due to the nature of the loops. It actually works as:  
for a in range(x+1):  
 for b in range(y+1):  
 for c in range(z+1):  
 if a + b + c != n:  
 print(stuff is here)

We start at [0,0,0]. Then c will increment to get to [0,0,1] When c hits [0,0,z], we get [0,1,0] as the next in the loop. This pattern continues and gives us the lexographical ordering required of the output. Inner loop executes first.

Elements defined inside List comprehension have a different scope, you can access elements at the outside in the list comprehension, but you cannot access elements defined in the list comprehension outside.

If an object contains a reference to itself, it's called a cyclic object. Python prints a [...] whenever it detects a cycle in the object, rather than getting stuck in an infinite loop. the three dots represent a cycle in the object.

Assignment Creates References, Not Copies. To avoid references and create copies use slicing operator L[:]

\*-operator to unpack the arguments out of a list or tuple

methods like insert, remove or sort that only modify the list have no return value printed – they return the default None.

Difference between range and xrange – range generates a list, xrange lazily generates values item by item. This is the difference in python2. In python3, range works like xrange, so no xrange in python 3.

For code efficiency , first filter then sort, because sort takes more memory than filter.

Whenever you assign a value to a variable, that value is stored in an object. That variable is a reference to the object. If you do list1=list2. List1 and list2 are referring to the same object and changing one reference, will change the object, hence the other reference is also changed. But if you give an entirely different value to the reference , this concept will not work. Only if you modify the list, either do insert, append, then this concept is applicable.

To access any element in the list we do – x[y]. This in turn calls the \_\_getitem\_\_ method of list.

If we assign a value to an index , x[y]=1. It is \_\_setitem\_\_

l1+=l2 #+= is \_\_iadd\_\_, \*= is \_\_imul\_\_

list multiplication list\*2– only adding the lists twice, We can multiply a list only with int, not with another list.

# Sets:

Sets and lists difference:

1. sets have no duplicate values, even if you define some, they will be removed
2. Sets are unordered,(i.e. no Indexing is possible)
3. Sets are mutable, but it allows only immutable objects inside it.
4. Mutable objects are not allowed inside set.

Initialising sets:

S=set()

S={} is wrong, S becomes empty dict

S={a,b}

One element set S={a} #No need to use comma as in tuple

S=set(l1) #Creating set from list, if list has duplicate entries, the set removes them.

To add elements – S.add(‘a’)

s.update(i) = this works only when i is an iterable object, like lists,strings etc. It wont work for int. Adds the elements of i to the set. Splits the elements of the iterable and adds it to the set.

s.discard(value)  
s.remove(value)

Both the discard() and remove() functions take a single value as an argument and removes that value from the set. If that value is not present, discard() does nothing, but remove() will raise a KeyError exception.

a.union(b) # Values which exist in a or b

a.intersection(b) # Values which exist in a and b

a.difference(b) # Values which exist in a but not in b

a.symmetric\_difference(b) #The term symmetric difference indicates those values that exist in either or but do not exist in both.

We cannot sort a set, to get the sorted values, convert the set to a list

'set' object does not support item assignment

# **Tuple**

A tuple consists of a number of values separated by commas.

Difference between lists and tuples are:

1. Lists are mutable, while tuples are immutable. Tuples are read-only lists.
2. Tuples are faster than lists.
3. If you know that some data doesn't have to be changed, you should use tuples instead of lists, because this protects your data against accidental changes.
4. Tuples can be used as keys in dictionaries, while lists can't.
5. As tuple cant be changed, they are non hackable. Good for security purpose.

tup1 = (50,)

for single element also, use comma.

You cannot remove single tuple element, you have to delete entire tuple. Whereas you can remove single element of a list, as list is just array of references.

del tuple\_name

cmp(tuple\_1,tuple\_2)

If tuple\_1 is smaller, -1 is returned, if it is greater, 1 is returned.

Nested tuples and nested lists are also possible.

B=[[1,2,],[3,4]] #Nested list

B[0] =[1, 2] , B[0][1]=2 #Same for tuples as well.

b = ("Bob", 19, "CS") # tuple packing

(name, age, studies) = b # tuple unpacking

>>> name

'Bob'

You can unpack all the datatypes - string, list, tuple

You have to assign a,b,c and so on values to the datatype based on the number of entries they have.

If you do not know how many return values are there for a function, use tuple to assign as the return value of function.

Tuples can be passed as keys to dictionary, not just tuples – specifically non mutable tuples. Because t= (1,2,[3,4]). t(2) is a list and it can be changed. This tuple t cannot be passed as a dict key. A tuple can be reliably hashed to a single value.

t=(t1,t2,t3) #This is tuple packing.

x, y, z = t #This is sequence unpacking. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence.

Note that multiple assignment - a,b,c=1,2,3 is really just a combination of tuple packing and sequence unpacking.

name = "John"  
age = 23

print "%s is %d years old." % (name, age)

To use two or more argument specifiers, use a tuple (parentheses):

# This prints out "John is 23 years old."

**Argument specifiers:**

%s – String, list, tuple (or any object with a string representation, like numbers)  
%d – decimal  
%f - Floating point numbers  
%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.  
%x/%X - Integers in hex representation (lowercase/uppercase)

The best ways is to use ‘format’ instead of argument specifiers.

Named tuples are the normal tuples, the only difference is that the tuple has a name. from collections import namedtuple.

from collections import OrderedDict – Ordered dictionary.

# Dictionary

Hash table type, consists of key-value pairs. Keys should be immutable types. Keys can be duplicate but if they are, the last value of key overwrites the first value.

Dictionary objects are enclosed by flower braces { } and values can be assigned and accessed using square braces.In perl we have name and value, here it is key and value.

Dictionary values can be of any type. But keys have some restrictions

Dicts are unordered set of key: value pairs, i.e they cannot be accessed by indices, they can be accessed only by keys. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys. Dictionary doesn’t follow any sequence, hence faster access.

Strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

Eg-   
plants = { "carrot": 1, "beetroot": 2 } #initialise a dictionary, using {}

plants["radish"] = 2 #add a key-value pair to the dictionary

print(plants["radish"]) #print the pair’s value, access using []

print(plants.get("tuna"," Not Found")) #If the pair is found, print the value, else print Not Found

plants["radish"] is similar to plants.get(“radish”)  
Difference is plants[“radish”] raises a key error if the key is not present, .get() do not raise a key error.

**d.get(“key\_name”,”default output if key not present”)**

You can access the keys and values using:

**keys = d.keys()**

**values = d.values()**

**dict.items() = key,value pairs**

# To Sort the keys from the dictionary, using the builtin sorted function.

keys = sorted(d.keys()) #sorted gives list type as output

Dictionary – “Key” and “Value”

**del d ['Key\_name']** – deletes the key ‘Name’ of the dictionary

**del d** – deletes the whole dictionary

**d.clear()** removes all entries in a dictionary, dictionary object remains as is.

Cmp(dict1,dict2) returns 0 if both dictionaries are equal, -1 if dict1 < dict2, and 1 if dict1 > dic2.

**d2=d1.copy()**

**d1.has\_key('Key\_Name')**   
gives 1 if ‘Name’ is present in the dictionary, if not 0

**dict.setdefault(‘key\_name’,value\_name)** = is similar to get, if the key exists in the dictionary, else, it will add a new key , key\_name and set its value to default. If value\_name is given it assigns that value to key\_name, else it gives default value None.   
If the key is already existing in the dictionary, it would not change its value. If the key is not present in the dictionary, that key-value pair is added to the dictionary. The return value of this command is the value for the key-key\_name

**d.update(d2)**

Adds dictionary dict2's key-values pairs to dict

Dictionary instead of +,use update, because \_\_add\_\_ is not defined for dict class, whereas it is defined for all other data types.

Tuple - ( )- Normal

List – [ ]- Sqaure

Dictionary – { }- Flower

The dict() constructor builds dictionaries directly from sequences of key-value pairs:

dict([('s', 4), ('g', 1), (j, 2)])

dict(s=4, g=1, j=2)

dict comprehensions

{x: x\*\*2 for x in (2, 4, 6)}

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

for k, v in d.items():

print(k, v)

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

for i, v in enumerate(['tic', 'tac', 'toe']):

print(i, v)

#i for index, v for value

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

for q, a in zip(L1, L2):

print('What is your {0}? It is {1}.'.format(q, a))

Another example of zip:

xvec = [10, 20, 30]

yvec = [7, 5, 3]

sum(x\*y for x,y in zip(xvec, yvec))

is sum of 70,100,90

“1.0” is same as “1” for dictionary key values.

Dictionaries are implemented with a hash table. the built-in hash function returns the same hash for equivalent numeric values, no matter if those values are int or float.

Dictionaries – are best suited when the data is labelled, i.e., the data is a record with field names.

lists – are better option to store collections of un-labelled items say all the files and sub directories in a folder.

Generally Search operation on dictionary object is faster than searching a list object.

# Operators:

Only \* and + operator can be used on string.

\*\* Exponent

// Floor division = gives integer value for quotient

^ = Binary XOR.

<< Binary Left Shift

>> Binary right shift

0b1100>>1 gives 6 as the output, it means that if we right shift 0b1100 by 1 bit , it results in the binary version of 6.

& = Binary AND

and = logical AND

if ( a in list ):

if value of a is present in the list, it is true.

Sum() function works for sets and lists as well

You can not use int() to convert a list to numeric, but map can achieve it

Identity operators compare the memory locations of two objects.

There are two Identity operators. is and is not.

a=20, b=20. a is b is true. a is not b is false.

If there are two multiplications, one on the left and one on the right. The left multiplication has precedence over the right.

round() function gives the nearest integer value. It is applicable on numbers.

AND, OR are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined.

AND has higher precedence than OR. If x or y and z is there, ‘y and z’ will be evaluated first, then the result of ‘y and z’ will be OR’ed with x. NOT>AND.

Difference between is and equals(==):

“is” checks if both the variables point to the same object whereas the == sign checks if the values for the two variables are the same. So if the is operator returns True then the equality is definitely True, but the opposite may or may not be True.

the reference implementation of Python caches integer objects in the range -5..256 as singleton instances for performance reasons. which means they are always the same single object and there is only one copy of each number.

When a new string is created, the Python interpreter can choose whether or not to store a cached copy of that string. This is called interning.

print 0.1 + 0.2 == 0.3  
prints False

print 1 + 2 == 3

prints True

because we cannot represent 0.1,02 accurately in binary.

If two strings are same, is returns True

If two lists are same, is returns False. Same for tuples and other datatypes.

Even if two numbers which are bigger are same, their ids may be different.  
Small numbers return True because of integer catching( interning). Python stores the address of the integer and assign the same id to a new object created.

~x is equivalent to -(x+1).

~ stands for bitwise inversion.

Hash() in python:

A hash is a fixed sized integer that identifies a particular value. Each value need to have it's own hash, they enable quick look-up of values in a large collection of values. Examples is in Python's set and dict. In a list, if you want to check if a value is in the list, with if x in values:, Python needs to go through the whole list and compare x with each value in the list values. This can take a long time for a long list. In a set, Python keeps track of each hash, and when you do - if x in values:, Python will get the hash-value for x, look that up in an internal structure and then only compare x with the values that have the same hash as x.

Every immutable type in python has a hash value, if two immutable types have the same value, their hash values are same. This makes lookup in set and dict very fast, while lookup in list is slow.

Moreover in sets, order doesn’t matter because python doesn’t keep track of the order, instead it keeps track of the hash value.

Hash – is a dictionary storage for values in python. Hash values should not be stored permanently, for security reasons. comparing two hashes is easier than comparing two objects, this is the use of hash. mutable objects are usually not hashable.

Hashable --> Immutable --> Used as keys to dictionary.

In python 2 – 5/2 is 2, in python3, 5/2 is 2.5  
// always gives integer result in both the versions.

Builtin function \_\_div\_\_ is replaced by \_\_truediv\_\_ in python3. For // use \_\_floordiv\_\_

BODMAS – Braces, Orders(powers, roots), division, multiplication, addition, subtraction.

If both division and multiplication are present, go from left to right.

# Loops:

break - to exit a for loop or a while loop.

continue - to skip the rest of the loop and go for the next iteration, i.e return to the "for" or "while" statement.

pass is a dummy statement – say for counting time.

‘IF’ cannot be treated as a loop when a break is encountered. ‘if’ is just a condition, only ‘for’ and ‘while’ are loops.

‘WHILE’ conditions are true for any non-zero values, only zero is considered to be false.

‘FOR’ iterates over sequence, sequence could be string, list, dict etc.   
In dict, for loop iterates over keys. To iterate over both keys and values of dict, use dict.items.

As int is not a sequence type, to convert to sequence use range.

While loop also can use corresponding else, just like IF. It is used when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

While loop has two purposes-

To check a condition that may change during the course of the loop, so that the loop can finish executing once the condition evaluates to false.

If that condition is too complicated to distil into a single expression, it makes more sense to keep the while expression always true and end the loop using a break statement.

print('r' if x<10 else 'w')

if else can be written in a single statement as well.

# Memory Management:

Python's memory allocation and deallocation method is automatic. Python uses two strategies for memory allocation reference counting and garbage collection.

1.Reference counting - Count the number of times an object is referenced by other objects in the system. When the reference count becomes zero the object is deallocated. Problem – if the object refers to itself, like in a recursive function, that object’s memory is not freed at all unless garbage collection is invoked. This is called reference cycle. A reference cycle is when there is no way to reach an object but its reference count is still greater than zero.

def make\_cycle():  
 l = [ ]  
 l.append(l)

make\_cycle()

2.Automatic Garbage Collection of Cycles- Python schedules garbage collection based upon a threshold of object allocations and object deallocations. When the number of allocations minus the number of deallocations are greater than the threshold number, the garbage collector is run. One can inspect the threshold for new objects (objects in Python known as generation 0 objects) by loading the gc module and asking for garbage collection thresholds:

import gc  
gc.get\_threshold()

Garbage collection thresholds: (700, 10, 10)

default threshold here is 700. This means when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.

Problem - Automatic garbage collection will not run if device is running out of memory, instead your application will throw exceptions, which must be handled or your application crashes.   
It places high weight upon the NUMBER of free objects, not on how large they are. If you want to free large blocks of memory use manual garbage collection.

Manual Garbage Collection

This could effectively handle memory being consumed by reference cycles.  
import gc  
gc.collect()

gc.collect() returns the number of objects it has collected and deallocated. Gives the number of reference cycles.

If we create a few cycles, we can see manual collection work:

import sys, gc

def make\_cycle():

l = { }

l[0] = l

def main():

collected = gc.collect()

print "Garbage collector: collected %d objects." % (collected)

print "Creating cycles..."

for i in range(10):

make\_cycle()

collected = gc.collect()

print "Garbage collector: collected %d objects." % (collected)

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

ret = main()

sys.exit(ret)

2 types of manual garbage collection: time-based and event-based garbage collection. Time-based: the garbage collector is called on a fixed time interval. Event-based: calls the garbage collector on an event. For example, when a user disconnects from the application or when the application is known to enter an idle state.

Do not run garbage collection too freely, as it can take considerable time to evaluate every memory object within a large system.

Run manual garbage collection after your application has completed start up and moves into steady-state operation. This frees potentially huge blocks of memory used to open and parse file, to build and modify object lists, and even code modules never to be used again.

Consider manually running garbage collection either before or after timing-critical sections of code to prevent garbage collection from disturbing the timing.

Heap Contains Remaining Memory

Stack Contains local variables

Global Contains global variable

Code Contains general instructions

Chr(x) – int to ascii. For string it doesn’t work

Ord(x) – char to int –reverse

# Python Memory Model:

A memory model is a structured way of representing variables and data in a program.   
  
every piece of data is stored in a Python program in an object. But how are the objects themselves stored? Every computer program (whether written in Python or some other language) stores data in computer memory, which you can think of as a very long list of storage locations. Each storage location is labelled with a unique memory address. In Python, every object we use is stored in computer memory at a particular location, and it is the responsibility of the Python interpreter to keep track of which objects are stored at which memory locations.  
Every object in Python has three important properties—id, value, and type

In Python, a variable is not an object and so does not actually store data; variables store an id that refers to an object that stores data. We also say that variables contain the id of an object. Python has object based memory model.

Two types of memory in python, Stack Memory and Heap Memory.

Function calls are stored in stack memory, all local variables initialisation is also in the stack memory. Stack represents the execution stack. Stack frame holds function calls and its associated local variables, A stack frame is added to the stack when a function is called.  
Heap memory is allocated during execution of instructions written by programmers. The variables needed outside of method or function calls or are shared within multiple functions globally are stored in Heap memory. Heap is elastic in size and holds the pointers to objects.  
In Golang stack size is dynamic and it borrows memory from heap. There is a stack per goroutine and main function is also goroutine, but because stack is dynamic, thousands of goroutines can be spawned. Only limitation being heap space in the system.

In Python memory management involves a private heap containing all Python objects and data structures. Python Interpreter takes care of this heap and programmer has no access to it.

# Built-in functions:

filter(function, sequence):  
Returns a sequence consisting of those items from the sequence for which function(item) is true.

From the sequence, it filters out, what values hold true for the function and returns them.

map(function, sequence):  
Calls function(item) for each of the sequence’s items and returns a list of the return values.

Map is a higher order function, that is – it is a function and it accepts another function as its input.

For each item in sequence, checks if the function is true or false and returns 1 or 0 based on the check.

list(map(int,[-1.0,-2.3,0,1,2.5]))

will return - [-1, -2, 0, 1, 2]

Difference between map and filter is that, map applies the function on the sequence and returns the result, whereas filter return the values in the sequence, for which the function is true.

reduce(function, sequence) returns a single value constructed by calling the binary function on the first two items of the sequence, then on the result and the next item, and so on.

If the function has 2 arguments, 2 sequences can be passed.

ZIP- This function takes two equal-length collections, and merges them together in pairs.

If the argument sequences are of unequal lengths, then the returned list is truncated to the length of the shortest argument sequence.

Zip(\*X) unpacks the list so zip() gets passed each element of X as an argument.

When you zip() together three lists containing 20 elements each, the result has twenty elements. Each element is a three-tuple.

n,m =(input().split())

input is 5 3. At a time, we cannot convert both n and m to int. So here map comes very handy.

N, M = map(int, input().split())

Sep:

print('foo', 'bar', sep=' -> ')

foo -> bar

sep needs minimum two arguments.

\* turns each element of a list into multiple arguments. print() accepts multiple arguments and separates them by space by default.

This is called unpacking.

\* iterable unpacking operator and \*\* dictionary unpacking operators.

\*a = range(5) #Invalid

\*a, = range(5) #Valid

Starred expressions are only allowed as assignment targets, using them anywhere else (except for star-args in function calls, of course) is an error.

>>> [\*range(4), 4]

[0, 1, 2, 3, 4]

>>> {'x': 1, \*\*{'y': 2}}

{'x': 1, 'y': 2}

Any returns True if any element of the iterable is true.

All returns True if all of the elements of the iterable are true.

any([1>0,1==0,1<0])

True

all(['a'<'b','b'<'c'])

True

import time

print(time.time()) at the beginning and at the end to mark the time recorded to perform the operation.

Difference between eval and exec is that eval is for a single statement, exec is for a piece of code.

eval(‘string\_name’) takes the string as code  
eval(‘2+3’) gives 5.

1. abs() #absolute value
2. all,any,ascii,bin,bool,complex
3. callable() – if the object has \_\_call\_\_ method(like functions, classes etc)
4. delattr(object, name) #deletes attribute object.name
5. dir() #returns list of names in the current local scope
6. enumerate(iterable, start=0) #gives object, you can convert it to list or tuple.
7. eval(expression, globals=None, locals=None)

|  |  | Built-in Functions |  |  |
| --- | --- | --- | --- | --- |
| abs() | divmod() | input() | open() | staticmethod() |
| all() | enumerate() | int() | ord() | str() |
| any() | eval() | isinstance() | pow() | sum() |
| basestring() | execfile() | issubclass() | print() | super() |
| bin() | file() | iter() | property() | tuple() |
| bool() | filter() | len() | range() | type() |
| bytearray() | float() | list() | raw\_input() | unichr() |
| callable() | format() | locals() | reduce() | unicode() |
| chr() | frozenset() | long() | reload() | vars() |
| classmethod() | getattr() | map() | repr() | xrange() |
| cmp() | globals() | max() | reversed() | zip() |
| compile() | hasattr() | memoryview() | round() | \_\_import\_\_() |
| complex() | hash() | min() | set() |  |
| delattr() | help() | next() | setattr() |  |
| dict() | hex() | object() | slice() |  |
| dir() | id() | oct() | sorted() |  |

dir() gives the list of attributes for the current module, list of imported modules,

## Format in python:

'{1} {0}'.format('one', 'two')  
Output- two one

'{} {}'.format(1, 2)  
Output- 1 2

#To keep the order intact, there is no need of numbering, if the order varies, then use numbering starting from 0.

# FUNCTIONS:

Lambda expression:lambda argument\_list: expression

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name. lambda is anonymous by default, if needed, we can give it a name.

The following example of a lambda function returns the sum of its two arguments:  
f = lambda x, y : x + y  
f(1,1)

Prints 2

Return value of a function is stored at function(x). if you assign , say z to function(x) and print z, the return value of the function is printed.

**Function Arguments:**

You can call a function by using the following types of formal arguments:

Required arguments:

passed to a function in correct positional order. Order would remain same

Keyword arguments:

def printinfo( name, age ): #Function Def

printinfo( age=50, name="miki" ) #Here we are passing 50 and “miki”. So we need not maintain the order of name, age – i.e. passing miki first and age next. Instead if we use keyword arguments (age, name) – you can pass those values in any order. We can associate those values with the parameter names.

Default arguments:

In the function definition you can set a default value for a parameter. In the function call, if you don’t set any value for that parameter, it takes the default value. If you set a specific value for that parameter in the function call, it takes that value (default value won’t be used).

Variable-length arguments:

An asterisk (\*) is placed before these arguments. This remains empty if no additional arguments are specified during the function call. This indicates that those variables are stored in tuple. If \*\* is used, they are stored in dictionary. We can use variable length arguments, \*only once. You cannot pass two variable length arguments at a time.

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable. It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the function. If we pass a list to a function, we have to consider two cases: Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

Perfect example :

def f(value, values):  
 value = 1  
 values[0] = 44

t = 3  
v = [1, 2, 3]

f(t, v)  
print(t, v[0])  
#The output is 3 44

This also shows that python is neither call by reference nor call by object, it is call by object reference. Because when you pass integer, that gets passed by value (that is the integer value doesn’t get changed, because only value is passed), when list is passed, it gets passed by reference(that is, the list value gets changed)

**Anonymous functions:**

They use lambda expression. These functions don’t use the standard function definition def.

sum = lambda arg1, arg2: arg1 + arg2

After defining function, the first string of the function is doc string-“”” “””.

To print the doc string - print(my\_function.\_\_doc\_\_)

Difference between definitions and methods:

Any normal function defined in python is called definition (we are using the def keyword)

If that function is defined inside a class, it is called method.

Only if you assign a value to a variable inside a function, it will become local. If you don’t assign value to that variable inside that function and it is already defined outside the function, it remains to be a global variable.

The execution of a function introduces a new symbol table used for the local variables of the function. All variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

Local—Enclosed--Global—Built-in LEGB

1.Default value arguments   
 #these are not mandatory  
 #Default arguments are evaluated when the function is defined, not when it is executed. Up to the time of fn definition, whatever value is defined, that value only is taken.

2.

Difference between Keyword arguments and default arguments:

def sortwords(\*wordlist, case\_sensitive=False):

This function accepts any number of positional arguments, and it also accepts a keyword option called 'case\_sensitive'. This option will never be filled in by a positional argument, but must be explicitly specified by name ‘case\_sensitive=’.

There is diff between arguments and parameters:   
functions have associated parameters,   
and function calls have arguments (fn(argument)).

Function definition = parameters  
Function call = arguments

“keyword argument” is, by definition, an argument with a keyword specified. And there’s no such thing as a “keyword parameter” in Python Any named parameter may be passed as either a keyword argument or a positional argument. These calls are all equivalent:

def fn(a, b=2):

return a + b

fn(1, 2)

fn(1, b=2)

fn(a=1, b=2)

fn(\*[1, 2])

fn(1, \*[2])

fn(\*[1], b=2)

fn(b=2, \*[1])

fn(1, \*\*{'b': 2})

fn(\*[1], \*\*{'b': 2})

fn(a=1, \*\*{'b': 2})

fn(b=2, \*\*{'a': 1})

fn(\*\*{'a': 1, 'b': 2})

It makes no difference here that the parameter b was specified with a default value b=2. All that does is give the caller the additional ability to omit the second argument.

use CamelCase for classes and lower\_case\_with\_underscores for functions and methods.

\* operator significance - you pass or you don’t pass, you have the privilege for that

The first issue is that the variable arguments are always turned into a tuple before they are passed to your function. This means that if the caller of your function uses the \* operator on a generator, it will be iterated until it’s exhausted. The resulting tuple will include every value from the generator, which could consume a lot of memory and cause your program to crash.

Functions that accept \*args are best for situations where you know the number of inputs in the argument list will be reasonably small. It’s ideal for function calls that pass many literals or variable names together.

The second issue with \*args is that you can’t add new positional arguments to your function in the future without migrating every caller.

Benefits of keyword arguments:

1. keyword arguments make the function call clearer to new readers of the code.

2. default values are specified in the function definition. This allows a function to provide additional capabilities when you need them but lets you accept the default behaviour most of the time. This can eliminate repetitive code and reduce noise.

Significance of None over declaring empty string or list etc. If u declare empty string, whenever you modify that value, the same string will be modified. Because the string is shared by all function calls, modifying one will modify the other.

Functions without return statements, also return a value which is None.

While using mutable objects for default arguments, instead of giving them mutable objects, give them None. By assigning None to default args, we say that no argument is provided, so hence for each function call

\*args = variable number of arguments

\*kwargs = variable number of keyword arguments

\*args : tuple : positional arguments  
\*kwargs : dictionary : keyworded arguments

\* is also used for unpacking the datatypes like lists, tuples etc

Inside a function header:

\* collects all the positional arguments in a tuple

\*\* collects all the keyword arguments in a dictionary

In a function call:

\* unpacks an list or tuple into position arguments

\*\* unpacks an dictionary into keyword arguments

All the following calls would be invalid,if :

If required argument missing

If non-keyword argument comes after a keyword argument

If duplicate value for the same argument

If unknown keyword argument

Order is required arguments, \*args, \*kwargs.

\*kwargs are passed as a=1,b=2,c=3.

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). The function definition does not execute the function body; this gets executed only when the function is called.

Arguments are passed using call by value (where the value is always an object reference, not the value of the object).

After passing the variable length arguments, you cannot pass positional arguments, though you can pass keyworded arguments, so that the variable length arguments can be separated from the next set of arguments using the keyword.

Positional and keyword arguments can be arbitrarily combined:

print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',

other='Georg'))

to round off to three places after the decimal.

‘{0:.3f}.'.format(math.pi))

If inside a function, we define a variable as global, whenever that function is called, that global variable assumes the value defined in that function.

Function definition has no meaning until it is called, once it is called all the function’s global variables are defined.

If you define a function def f():  
the type of the function is the return value of the function. If f() returns None, type(f()) is None type.

Recursive functions should and have to “TERMINATE”. Please make sure to terminate them.

**Closure in python:**

Closure in Python when a nested function references a value in its enclosing scope.

The criteria for closure :

We must have a nested function (function inside a function).  
The nested function must refer to a value defined in the enclosing function.  
The enclosing function must return the nested function.

Closures can avoid the use of global values and provides some form of data hiding.

The \_\_closure\_\_ attribute returns a tuple of cell objects which contain details of the variables defined in the enclosing scope. Python’s late binding behaviour which says that the values of variables used in closures are looked up at the time the inner function is called.

Difference between closures and nested functions:

Objects are data with methods attached, closures are functions with data attached.

Sometimes we want a function to retain a value when it was created even though the scope cease to exist. This technique of using the values of outer parameters within a dynamic function is another way of defining the closure.

Practical uses of closures:

Replacing hard coded constants

Eliminating global

Methods in class are also kind of closures. You create a class object, class is the outer function, from the class object, you call the method, method is the inner function.

Print function definition looks like:

def print(\*objects, sep=None, end=None, file=None, flush=False):

if sep is None: sep = ' '

if end is None: end = '\n'

if file is None: file = sys.stdout

file.write(sep.join(map(str, objects)) + end)

if flush: file.flush()

# MODULES:

A module is a Python object with arbitrarily named attributes that you can bind and reference.

List of modules:

* Csv
* datetime
* Ftplib
* gc
* Getopt - parse command-line options and arguments.
* Glob – filename pattern matching.
* logging
* math
* Os - interacting with the operating system
* Re
* Shutil - file and directory management tasks
* signal
* subprocess – spawn process and connect to I/O stream
* Sys – interacting with interpreter.
* Telnetlib
* Textwrap
* Threading
* Time
* Traceback
* Weakref
* Dis – disassembler of python byte code to mnemonics
* cProfile

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module. A directory of such files is a package.

Directory = Package

File = Module

Difference between import os and from os import \*. This will keep os.open() from shadowing the built-in open() function which operates much differently.

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user’s global variables.

globals()-returns a dictionary of variable names to variable values, where the variable names are considered global in scope at that moment.

import module\_name

content = dir(module\_name)

help(str) will give all the methods in string.

C:\Users\lapoorva\AppData\Local\Programs\Python\Python35-32\Lib – At this path , you see all the library functions and their methods

print content – prints sorted list of strings containing the names defined by a module.

To speed up loading modules, Python caches the compiled version of each module in the \_\_pycache\_\_ directory

Python checks the modification date of the source against the compiled version to see if it’s out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

A program doesn’t run any faster when it is read from a .pyc file than when it is read from a .py file; the only thing that’s faster about .pyc files is the speed with which they are loaded. The .pyc file is the compiled bytecode, which is then interpreted. Only loading speed is increased, run time is not different.

Import package,subpackage.module

While using ,. syntax, only the last item could be a module, all others have to be packages(or subpackages).

os.rename( "test1.txt", "test2.txt" )

os.remove("text2.txt")

os.mkdir("dir\_name")

os.makedirs(whole path)

os.getcwd()

os.rmdir('dirname')  
#removes only empty directory.  
to remove non-empty directory, shutil.rmtree()

shutil.copyfile(file1, file2)

shutil.move(file1, file2f)

dis.dis(function\_name)

Pythons’ debugger’s prompt is (Pdb)

!a

p a

post mortem should be in except mode. It will be executed only when there is exception.

Pdb – track with the arrow mark ->

TO create a package:

Make a directory

Give a file \_\_init\_\_.py = to indicate that this is a python package, not a normal directory. You can control the loading of modules, you can specify which module to be loaded, by from module import \*

If your environment is Windows – import sys. Sys.path.

If it is unix, it is in PYTHONPATH

import glob

glob.glob('\*.py') – is similar to ls , will list all .py files

cProfile.run(‘f(x)’) #gives the number of calls to the function f.

Monkey patching: You have a class definition in source code, you cannot modify that, instead you chose to modify the methods of the class at run time by assigning the class methods to our user defined methods. Specially used in testing.

class C():

def \_\_init\_\_(self, value):

self.value = value

def cmethod(self):

print(self.value+"cmethod")

c=C(“hi”)

c.cmethod() #prints hicmethod

def monkmethod(self):

print(self.value+"monkmethod")

C.cmethod=monkmethod

d=C(“hi”)

d.cmethod() #prints himonkmethod.

**GLOB MODULE:**

To match the pattern for filename search.  
glob.glob(‘python\*’)

To match all the filenames starting with ‘python’

\*matches all the continuous characters, ? matches single character in that position.

[0-9] matches the characters in the range 0 to 9.

**TRACEBACK:**

traceback.print\_tb(traceback\_object)

traceback object is sys.exc\_info[2]

While importing modules, python has different paths for diff modules in diff versions of python. In the try block, try importing from a given path, and cath the ImportError exception in the except block and give a diff path to import in the same except block.

## SYS:

sys.getrefcount(variable\_name) #gives all references to that variable

sys.getsizeof(obj) #gives the size required by interpreter to hold that file

os.path.getsize(filepath\_filename) #gives the actual size of the file

sys.path #list having the search path for modules

sys.platform gives the OS name.

sys.modules gives the dictionary with the module names (in string) as keys and their paths as values.

## OS:

Shell variables os.environ

Running programs os.system, os.popen, os.execv, os.spawnv

Spawning processes os.fork, os.pipe, os.waitpid, os.kill

Descriptor files, locks os.open, os.read, os.write

File processing os.remove, os.rename, os.mkfifo, os.mkdir, os.rmdir

Administrative tools os.getcwd, os.chdir, os.chmod, os.getpid, os.listdir, os.access

Portability tools os.sep, os.pathsep, os.curdir, os.path.split, os.path.join

Pathname tools os.path.exists('path'), os.path.isdir('path'), os.path.getsize('path')

os.path.split(complete\_path\_name, file\_name)

os.path.name(just the path\_name, filename\_of\_the\_file\_tobe\_added\_to\_the path)

os.path.splitext(any\_path\_name)  
will convert the path name as specific to the OS.(\\ for windows, / for unix etc)

os.system is equivalent to subprocess.call

## Subprocess:

>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)  
>>> pipe.communicate()  
(b'The Meaning of Life\r\n', None)  
>>> pipe.returncode  
0

# EXCEPTION HANDLING:

The run time errors are exceptions. For the syntax errors, program fails at compilation. So, execution will not happen. After compilation, if you have some errors, those are exceptions. Syntax may be correct. Like 1/0 , but while u execute it, it gives an error. Syntax errors, also known as parsing errors

Syntax errors = parsing errors

If you access non-existing memory location, the script will terminate abruptly. Eg-Crash of web browser. Ungraceful terminations. Run-time errors

IndexError – accessing a defined list, with a non existing index

Name error – accessing undefined variables

Key error - unexisting dictionary key

ZeroDivisionError – 1/0

except error\_name:

i.e. name error,key error etc..

If there is an error – Type Error - takes 2 positional arguments but 3 were given. It means that the first number takes 2 positional arguments ==> it actually has 2 arguments defined, but while calling you have given 3 arguments which is wrong.

except block will catch the error and will not allow for termination and will change the direction of the program, so that crash will not happen.

Try block can have both except and finally.

If try block is not throwing a runtime error, else block will execute.

difference between exception and error in python:

Error at runtime is exception.

difference between exception and assert in python?

Exceptions should be used for errors that can conceivably happen, and you should almost always create your own Exception classes.

Asserts should be used to test conditions that should never happen. The purpose is to crash early in the case of a corrupt program state. Assertions are a systematic way to check that the internal state of a program is as the programmer expected, with the goal of catching bugs. In particular, they're good for catching false assumptions that were made while writing the code. Eg- If assert condition is False, it gives AssertionError.

An assertion is a condition that we’re claiming should be true at this point in the program. Typically, it summarizes the state of the program’s variables. Assertions review what has happened so far in the program, and show that the code we have written till now has the desired effect.

Assert syntax:

assert condition,statement to be printed.

Eg – assert 1==0, “sorry,one is not 0”  
Please note that only if the condition is false, the statement to be printed appears on the screen. That statement is nothing but the error information.

assert condition is equivalent to

if not condition:

raise AssertionError()

To catch all the exceptions –

except Exception as e:

print("Error is:\t\t ",e)

If no exception is raised, the else-block is executed.

No matter what happened previously, the final-block is executed once the code block is complete and any raised exceptions handled. Even if there's an error in an exception handler or the else-block and a new exception is raised, the code in the final-block is still run.

assert is intended for cases where the programmer calling the function made a mistake, as opposed to the user. Using assert under that circumstance allows you to make sure a programmer is using your function correctly during testing, but then strip it out in production.

Its value is somewhat limited since you have to ensure you exercise that path through the code, and you often want to additionally handle the problem with a separate if statement in production. assert is most useful in situations like, "I want to helpfully work around this problem if a user hits it, but if a developer hits it, I want it to crash hard so he will fix the code that calls this function incorrectly."

assert is similar to throwing an exception if a given condition isn't true.

except Exception as inst:

print(type(inst))

print(inst.args)

inst.args will print the exception’s error message. Type(inst) will print the error type whether it is zerodivisionError, keyError etc.

to create own exceptions, create a new exception class and this class should inherit from the Exception class.

Finally is executed always, whether exception occurred or not. If exception is not handled by an except clause, finally is executed first and then the exception is raised. The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement. finally block, which gets executed every time the associated try, except and else blocks finish

Else block – it tries something, if try is successfully only, then it is executed. It is useful for code that must be executed if the try clause does not raise an exception. It is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try ... except statement.

raise NameError(“Wrong Name”)

Wrong Name is the is the argument(or attribute) for the exception, i.e.the error message which we get

Pass by value – a copy of the argument is created and any change in the original object will not have an impact on the copy

Pass by reference – a reference is being passed, and if the value at the reference is changed, our value also gets changed.

In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called names. Likewise, "assignment" is really the binding of a name to an object. Each binding has a scope that defines its visibility, usually the block in which the name originates. In Python, values live somewhere on their own, and have their own types, and variables are just names for those values. It's not the same as pass by value, because you're not copying values. It's not the same as pass by reference, because you're not making references to variables.

In pass by reference – you are passing the list itself, if you change the list, the values passed to the function also gets changed. The object id is not changed. In pass by value – say you are passing the output of –‘slicing operation on the list’. This output is stored in the temporary object id. Outside the function, the value of the list is not changed. Whereas in pass by reference, the value of the list is changed outside the function as well.

If tuple consists of lists, and if the lists are changed, the tuple also gets changed.

except block has exception handlers. If some exception occurs in try, exception handlers in the except block are searched for.

# FILE I/O:

Difference between raw\_input and inputs is – input evaluates the given expression, whereas raw\_input takes the string as it is, no evaluation(for python2 only).

In python 3 raw\_input changed to input() and the input() in python 2 can be emulated by eval(input()).

type(raw\_input()) in python2 and type(input()) in python3 is always string.

type(input()) in python2 and type(eval(input())) in python3 is always the type of the input entered.

input() in python2 does the things you didn’t want it to.

Any file manipulation needs to be done

file object = open(file\_name, [access\_mode],[buffering])

The file\_name argument is a string value

As soon as you do fo.read(), the cursor always goes to the end, to perform further operations, place back the cursor to the beginning.

access\_mode:r,r+(read+write- cannot truncate a file,w+ can truncate),a+,b is for binary mode, rt is for text mode.

SUMMARY:

If you see ‘r’ , the cursor is always at the begining.

If you see ‘w’, the cursor is always at the beginning, because w overwrites the existing file and starts from the beginning.

If you see ‘a’, the cursor is always at the end.

Adding + to both, just adds on the other operation to it (adds r to w and w to r and r to a). No difference in the cursor position.

Write = newfile, overwrite

Read = no newfile

Append = newfile, no overwrite

buffering: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

Difference between r+ and w+. If you open in r+ and write something, those contents will be written at the starting of the file,overwriting the already written lines, but only till the point u fill, it is not ‘truncate’, it is ‘overwrite’ . In w+ at the end of the file everything is truncated, you have extra privilege of reading as well.

file.closed, file.mode , file.name , file.softspace

Returns false if space explicitly required with print, true otherwise.

If you open a file in read mode and the file doesn’t exist, it throws an error. If you open in write mode, a new file will be created if the file doesnt exist.

**fo.write**(“Any String”)

**fo.writelines**(“Any List”)

If list has 10 items, those 10 items will be written in 10 lines of the file

**fo.close()**

**fo.read**(count\_of\_bytes)

1 char – 1 byte

**fo.tell()**

returns the current position of the file object

**fo.seek**(offset, from\_where)

**fo.name**

**fo.flush()**

flush the data, before closing a file

**fo. fileno()**

File descriptor is a low-level concept, it's an integer that represents an open file. Each open file is given a unique file descriptor.

In Unix, by convention, the three file descriptors 0, 1, and 2 represent standard input, standard output, and standard error, respectively.

>>> import sys

>>> sys.stdin.fileno()

0

>>> sys.stdout.fileno()

1

>>> sys.stderr.fileno()

2

0,1,2 are the default file numbers, whatever file descriptor you define the first, takes the number 3, the next file descriptor takes 4 etc.

To redirect the

Readlines = new line

Read = EOF

Help(file)

**string=f.readline()**

reads single line from the file

the next f.readline() will read the next line where the cursor is now positioned. The cursor in python is dynamic. Each command you execute will change the cursor position to next.

f.readline(5) will read only the initial first 5 bytes of that line.

If you want to ignore the first line do, f.readline() and then again do f.readline() and perform operations on the next line.

**list=f.readlines()**

reads all the lines of a file into a list of strings. List[0] will have the first line. When passing the size to readlines , it will read the whole line, if only memory is available to read the complete line. If memory is insufficient, it will not read the line.

Difference between read, readline, readlines.

open('file').read() # read entire file into string

open('file').read(N) # read next N bytes into string

open('file').readlines() # read entire file into line strings list

open('file').readline() # read next line, through '\n'

While writing to the file, don’t forget to enter \n. write will not take newline by default, unlike python which takes newline by default.

fo.writelines[list1]

will write list1 to the file

fo.read(n) reads from presently where the cursor is to - till n bytes after that cursor

os.path.isfile('tmp.txt')  
gives True or False

Difference between f.close() and f.closed is that .close() is a function call and f.closed is the object of the file class. In f.closed which returns boolean variable, we are accessing just the object of the file. In f.close() we are calling a function and that function will perform the action of closing the file

.pyc – compiled format, modules packages will be stored in compiled format. Which is not understandable by the end user. If you make changes in modules and packages, you have to recompile them.

.pyw – GUI based format.

Try: block and except: block

# Regular Expressions:

Pattern oriented searching and string based search.

re.match(pattern, string, flags=0)  
Difference between search and match is search looks anywhere in the string. Match looks only at the beginning.Match is derived from search, it is the form of ‘search at the beginning’.

Raw string, if we put r – the string is taken as it is, taken raw.

Print(r”\n”) will print \n, normal print would print a newline.

If your text t is “Python”. Using reg-ex – so=re.search(“(pyt)”,t,re.I)

so.group(0) = pattern which you are searching for

so.group(1) = the matched hit which is Pyt.

groups() group(0) group(1..n)

match.group(0) always returns the search pattern for our search

match.group(1) match.group(2), ... will return the capture

# groups in order from left to right in the input string

# match.group() is equivalent to match.group(0)

Difference between () and [] in re is that () is used for grouping, whereas [] is used to indicate a range. Grouping only works when () is present in the search pattern. [] represents a character class.

Re.search() stops after the first match, so this is best suited for testing a regular expression more than extracting data.

use re.findall() to perform a global search over the whole input string.

If you need additional context for each match, you can use re.finditer() which instead returns an iterator of re.MatchObjects to walk through.

\d Any Digit

\D Any Non-digit character

. Any Character

\. Period

[abc] Only a, b, or c

[^abc] Not a, b, nor c

[a-z] Characters a to z

[0-9] Numbers 0 to 9

\w Any Alphanumeric character

\W Any Non-alphanumeric character

{m} m Repetitions

{m,n} m to n Repetitions

0 is default m, end is default n

\* Zero or more repetitions

+ One or more repetitions

? Zero or one

\s Any Whitespace

\S Any Non-whitespace character

^…$ Starts and ends

(…) Capture Group

(a(bc)) Capture Sub-group

.\* Capture all

abc|def Matches abc or def

Wildcard . will match any character, to match period in specific you use \.

\* and + are greedy patterns. They match till infinity.

regex = r"([a-zA-Z]+) (\d+)"

re.sub(regex, r"\2 of \1", "June 24, August 9, Dec 12",count)

\2 is the second match, \1 is the first match.

if count is 0, all matches are replaced. If it is 1, only 1st match is replaced.

Regular expressions use ‘\’ as escape sequence, python also uses ‘\’ as escape character, so to write one \ in regex string – for regex use \\, for python for each \ use escape sequence \. So 2x2 – 4 backslashes in total.

(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline. Dotall is the flag.

ab\* matches a, ab, abbbbb etc. Zero or more

ab+ matches ab,abbbb etc. One or more

ab? Matches a,ab only Zero or one

Except for control characters, (+ ? . \* ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Regular Expressions work only on strings, they have data dependancy. Re search goes char by char, first match found, then the search terminates. Only re.search supports grouping. And grouping is done by parantheses.

\w includes all forms of characters that couldbe used in variable names. A-Z,a-z,0-9,\_ ..  
Space and comma also have a siginificance in re. They shouldn’t be used unnecessarily in the re pattern.

^ has two meanings. If used inside [] it is – negation. If used outside, it indicates the beginning of a string. Inside [] not is applied to all the entries in []. To write \w without \_ = [^\_\W]. Because you cannot do one not while using ^.

\*is for 0 or more occurances, that means, if the pattern is not present, it doesnt throw an error.

phone = "2004-959-559 # This is Phone Number"

num = re.sub(r'#.\*$', "", phone)

num will be '2004-959-559 '

Python(?=!)

Match "Python", if followed by an exclamation point.

Python(?!!)

Match "Python", if not followed by an exclamation point.

regex = r"([a-zA-Z])\1([0-9])\2"

will match (‘a1bb222c3dd444') [('b', '2'), ('d', '4')]

\1 is for repetitions, for the first repetition \1, for the second repetition \2 etc..

re.search(regex1, 'a1b22c333d4444').groups()

('22', '333')

re.search(regex2, 'a1b22c333d4444').groups()

('c', 'd')

regex1 = r"([\d]{2})[^ ]([\d]{3})"

regex2 = r"([a-z])\d{3}([a-z])"

regex = r"([aeiou])\1+" replacement = r"\1"

will replace the duplicate vowels with single vowel.

Special metacharacters don’t match themselves.

. ^ $ \* + ? { } [ ] \ | ( )

Use ^ and $ often to match the beginning and the ending of the string.

A groupdict() expression returns a dictionary containing all the named subgroups of the match, keyed by the subgroup name.

m=re.match(r'(?P<user>\w+)@(?P<website>\w+)\.(?P<extension>\w+)','myname@hackerrank.com')

>>> m.groupdict()

{'website': 'hackerrank', 'user': 'myname', 'extension': 'com'}

re.findall() returns all the non-overlapping matches of patterns in a string as a list of strings..

match = re.search(pattern1, text)

match.re.pattern and match.string gives search pattern and the string to be searched for.

finditer() returns an iterator that produces Match instances instead of the strings returned by findall().

for match in re.finditer(pattern, text):

s = match.start()

e = match.end()

print 'Found "%s" at %d:%d' % (text[s:e], s, e)

finditer plays with that object using .start and .end, but remember that object is iterable, so to use the .start and .end you have to use the loop.

Python extends the basic grouping syntax to add named groups. Using names to refer to groups makes it easier to modify the pattern over time, without having to also modify the code using the match results. To set the name of a group, use the syntax (P?<name>pattern).

Difference between \1 and {} is that \1 matches anywhere in the string, but {} only goes for consecutive matches.

\1 is known as back referencing.

re.search gives search object, whereas re.sub returns a string

re.sub(re\_pattern, replacement pattern, string to be replaced)

pattern=re.compile(regularExpression)

patter.search(string\_to\_be\_searched)

## Looking Ahead, or Behind

If you want to see if the parentheses are correct – i.e. Opening and closing braces are present or no braces are present, then use look ahead to see, if they are correct.

Look Ahead = ?=

Negative = ?!

Look Behind = ?<=

?= For example, Isaac (?=Asimov) will match 'Isaac ' only if it’s followed by 'Asimov'.   
?<=abc)def will find a match in abcdef, since the lookbehind will back up 3 characters and check if the contained pattern matches.

positive lookahead means that you want your pattern to be followed by a specific character/expression

negative lookahead means that you want your pattern to be followed by anything but certainly not by a specific character/expression

The reason behind using these, is that, it will not consume the matched characters, unlike the other regex matches, which consumes the characters it matched.

r"\n" is a two-character string containing '\' and 'n', while "\n" is a one-character string containing a newline.

"a|b" is same as r'[ab]' #a or b

No need to escape . in raw string format [], r’[]’ in such format.

match = re.search('(?P<name>.\*) (?P<phone>.\*)', 'John 123456')

match.group('name')  
prints 'John'

match.groupdict()  
prints {'name': 'John',’phone’: ‘123456'}

# Iterators and Generators:

For loop over iterable items:

String : characters

List : item

Dictionary : keys

File : lines

All these are iterable objects.

There are many functions which consume these iterables like FOR loop.

In File: lines, lines are of string type.

The built-in function iter takes an iterable object and returns an iterator.

Generators simplifies creation of iterators, Generators implement iterators. A generator is a function that produces a sequence of results instead of a single value. Yield returns sequence of values unlike return which returns single value. Yield sequence is not a complete sequence, it is iterated sequence.

Generator generates a sequence of values.

There are times, though, when it's beneficial to have the ability to create a "function" which, instead of simply returning a single value, is able to yield a series of values. To do so, such a function would need to be able to "save its work," so to speak. It is said, "yield a series of values" because our hypothetical function doesn't "return" in the normal sense. return implies that the function is returning control of execution to the point where the function was called. "Yield," however, implies that the transfer of control is temporary and voluntary, and our function expects to regain it in the future. Such functions are generators.

The return statement causes your function to exit and hand back a value to its caller. The point of functions in general is to take in inputs and return something. The return statement is used when a function is ready to return a value to its caller.

def foo():

print("hello from inside of foo")

return 1

Now you can run code that calls foo, like so:

if \_\_name\_\_ == '\_\_main\_\_':

print("going to call foo")

x = foo()

print("called foo")

print("foo returned " + str(x))

Output:

going to call foo

hello from inside foo

called foo

foo returned 1

The interpreter writes return values to the console so the confusion between print and return.

Where containers are typically finite, an iterable may just as well represent an infinite source of data.

>>> x = [1, 2, 3]

>>> y = iter(x)

>>> z = iter(x)

Here, x is the iterable, while y and z are two individual instances of an iterator, producing values from the iterable x. Both y and z hold state, as you can see from the example. In this example, x is a data structure (a list), but that is not a requirement. So what is an iterator then? It's a stateful helper object that will produce the next value when you call next() on it. Any object that has a \_\_next\_\_() method is therefore an iterator. There are two types of generators in Python: generator functions and generator expressions. A generator function is any function in which the keyword yield appears in its body. Generator expressions are generator equivalent of a list comprehension.

An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables. The iter() function (which in turn calls the \_\_iter\_\_() method) returns an iterator from them.

If we use iter on an iterable datatype(like list, dict etc..), this will generate an iterator. The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object.

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

Here is how a generator function differs from a normal function.

Generator function contains one or more yield statement.

When called, it returns an object (iterator) but does not start execution immediately. (Very important for saving memory)

Methods like \_\_iter\_\_() and \_\_next\_\_() are implemented automatically. So we can iterate through the items using next().

Once the function yields, the function is paused and the control is transferred to the caller.

Local variables and their states are remembered between successive calls.

Finally, when the function terminates, StopIteration is raised automatically on further calls.

The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time. They are kind of lazy- producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

A normal function to return a sequence will make the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large. Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

Working of the FOR loop is as follows:

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. You can call the \_\_next\_\_() method using the next() built-in function;

\_\_iter\_\_() and \_\_next\_\_() are collectively called the iterator protocol.

Difference between iter() and \_\_iter\_\_():

iter(iterableobject\_likeListsString\_etc) generates an iterator object, on this iterator object, you can use the function next().

Application of iter() is to read lines of a file until a certain line is reached. with open('mydata.txt') as fp:

for line in iter(fp.readline, ''):

print(line)

This reads a file until an empty string is reached.

An iterator object(like list, string etc) has an \_\_iter\_\_() method so that you can call iter() on it to get an iterator for it, even though it already is one.

iter(iterable) returns an iterator so that you may call next(iterator) to get the next item from the iterable.

Generators create iterators, iterators work on the already available iterable objects. In generators \_\_iter\_\_() and \_\_next\_\_() methods are created automatically. In generators local variables and execution state are automatically saved between calls. when generators terminate, they automatically raise StopIteration. 3 features - automatic method creation, saving program state, automatic StopIteration.

Generator expressions are in (), these are more memory friendly than equivalent list comprehensions.

For loops loop over iterables.   
iterables have \_\_iter\_\_ functions   
\_\_iter\_\_ functions return Iterators  
Iterators make use of the next method to move from element to element within their associated iterable  
Once an iterator runs out of things to return from the next function it raises the StopIteration exception whenever next is called.

def \_\_iter\_\_(self):

return self # because the object is both the iterable and the iterator

## Decorators:

Decorators allow you to make simple modifications to callable objects like functions, methods, or classes. a decorator is just another function which takes a functions and returns one.

The @ indicates the application of the decorator.

Functions and methods are called callable as they can be called. In fact, any object which implements the special method \_\_call\_\_() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable. Basically, a decorator takes in a function, adds some functionality and returns it.

Simple terms – decorator is a function, which takes in a function and returns some desired functionality.

Pass the fn\_tobe\_decorated to the decorator function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

fn\_tobe\_decorated = decorator\_fn(fn\_tobe\_decorated).

Instead of writing this, add @decorator\_fn before the definition of the fn\_tobe\_decorated.

If some operations are to be performed, decorator\_fn generally has an inner function defined, because on the outer decorator\_fn, the fn\_tobe\_decorated is passed as a parameter, to pass further parameters use the inner fn. And on the outer return , return the value of the inner fn.

functions are first class objects. What that means is once a function is defined in a scope it can be passed to functions, assigned to variables, even returned from functions. This simple fact is what makes python decorators possible.

When you do x=f(), assigning a value to the function, that value is the return value.

If return statement returns a function, i.e., return f1(f1 is another function), then x=f(), x can be called, like x().

Decorator takes in the fn

# Object Oriented Programming:

Objects oriented programming has following advantages:

1) Simplicity

2) Modifiability

3) Extensibility and Maintainability

4) Re-usability

5) Security

6) Polymorphism

EAIP – Encapsulation, Abstraction, Inheritance, Polymorphism

**Encapsulation** is implemented through classes. A class bundles data and methods into a single unit. In fact the private data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class. you can restrict access to methods and variables. This can prevent the data from being modified by accident. Like private methods and private variables. The encapsulation hides the implementation details of a class from other objects. Using encapsulation, we can hide an object’s internal representation from the outside. Encapsulation is a way to can restrict access to methods and variables from outside of class. Data Security and Information hiding are the advantages of encapsulation.  
Public Member – Accessible outside the class  
Protected Member – Accessible within the class and subclasses  
Private Member – Accessible only within the class  
  
**Abstraction** is a natural extension of encapsulation. In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years — with changes along the way — is difficult. Abstraction eases this problem. Applying abstraction means that each object should only expose a high-level mechanism for using it. Hide internal implementation details. It should only reveal operations relevant for the other objects. Abstraction means hiding internal details and showing functionality

If two classes are implementing something similar, it means that there is a layer of abstraction we are not utilizing.

Encapsulation and abstraction can help us develop and maintain a big codebase. To Reuse common logic use inheritance.

Types of inheritance:

Direct inheritance

Multi-level inheritance

Multiple inheritance

Diamond inheritance

Method Resolution Order:

In cases of multiple inheritance, the order in which base classes are searched when looking for a method is often called the Method Resolution Order.

diamond inheritance: B from A, C from A. D from (B,C). so the order is D,B,A,C,A.

Membership id. Pointer to current object.

If a class inherits both parent and super parent – error in method resolution order.

**Polymorphism**-   
Same name, Many forms – Poly + morph  
Same function can be used on multiple datatypes. It is the ability of an object to adapt the code to the type of the data it is processing. Polymorphism is implemented using function overloading, method overriding, and operator overloading

In python child class can define a method with same method name as that in the Parent class and have customised implementation. Example – If parent class has len method, child class can also have len method, with its own customised implementation. In golang, interfaces help in achieving polymorphism. Method of an interface can be called on any object which implements that interface.

Eg-len is a builtin function and you can give both list and string to it. Polymorphism has two major applications in an OOP language.   
The first - an object may provide different implementations of one of its methods depending on the type of the input parameters.

The second - code written for a given type may also be run on derived types., i.e. methods understand the class hierarchy of a type.

Runtime polymorphism and compile time polymorphism:

At compile time itself, we would know which function to be called.  
Method overloading(Operator overloading) - Compile time Polymorphism  
Method overriding - Runtime polymorphism  
  
method overriding works in combination with inheritance.

Method overloading – same method called with different number of parameters.

Compile time is static, runtime is dynamic

## Difference between object oriented programming and functional programming:

OOP - treats all the data as objects. Data and its behaviour is kept in the objects.  
Functional – data is separate and the behaviour is separate.

Data- Ingredients, Function – recipe

Functional programming takes the divide-and-conquer approach, where the problem is divided into sub-problem, then each sub-problem is solved (creating a function to solve that sub problem) and the results are combined to create the answer for the whole problem.

OOP mimics the real world by create a mini-world inside the computer with many objects, each of which has a unique characteristic, and interacts with others. From those interactions the result would emerge.

# **CLASS**:

\_\_init\_\_:

1. Variables for the class(Parameters passed to the class)
2. Memory allocated to the class
3. Initial value with which the class object begins

Python doesn’t have something called constructors and destructors. You cannot call init as a constructor , but the functionality is same as that of a constructor. Python calls this when we create a new instance of this class.

E.g.- in function definition, we define def fun(parameter\_name): but in class definition we cannot do the same, class classname(parameters\_name): For class definition, () indicates the parent classes it inherits.

Self:

1. Reference to current class object(instance).
2. Instance identifier. The statements within the methods have automatic access to the current instance attributes.

For functions (methods) in a class to use the parameters passed to the class, there is no need to pass those arguments in the function call as well. Just use those parameters as- self.p1, self.p2 etc.

Self= all parameters to be passed will fit in self. As soon as an object of a class is created,

For self, class object passes values, not the function object.

Self is not a keyword, it can be replaced by xyz, but the recommendation says to use self. It also implies that if we have a method which takes no arguments, then we still have to define the method to have a self-argument. it's often called self, since this parameter gives the object being created its identity. The first argument \_\_init\_\_() gets is used to refer to the instance object, and by convention, that argument is called self.

\_\_new\_\_ is defined internally for every class definition, so that whenever we do obj=class\_name(), it will allow us to create a new object. So \_\_new\_\_ is a kind of constructor, \_\_init\_\_ is just to initiate that newly created object with some initial values.

\_\_new\_\_, \_\_init\_\_, \_\_del\_\_ are called magic methods.

Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

class variables are internally handled as dictionaries. If a variable name is not found in the dictionary of the current class, the class hierarchy (i.e., its parent classes) is searched

Class is like template, say you have a 90degree protractor, using that protractor you can draw multiple 90degree angles, like blue, green, red etc. the object is the one which makes the class alive. These blue, green angles makes the protractor alive. Class doesn't have any actions to do, only functions defined inside the class does those actions.

2. attribute references- o.i, o.f etc...

Attribute names are methods and data attributes(data members)

The instantiation operation (“calling” a class object) creates an empty object.

x=MyClass()

You are calling the class object(Myclass()) and assigning it to instance object x.

objects are created with instances customized to a specific initial state. Therefore \_\_init\_\_. As soon as u define, \_\_init\_\_ in a class, whenever you instantiate a class object, \_\_init\_\_ is invoked. So arguments to be passed to the class are passed to init.

the special thing about methods is that the object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x) not to MyClass.f(), unlike data members where x.i is equivalent to MyClass.i .

Complex explanation- When an instance attribute is referenced that isn’t a data attribute(x.f not x.i is referenced, fn is referenced), its class is searched(inside the class definition search for fn definition). If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object(Attach the fn object found to the present instance object which makes it method object): this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.(Now the arguments to method object are the instance object itself + the arguments list passed, by instance object itself we mean self).

instance variables are for data unique to each instance object and class variables are for attributes and methods shared by all instances of the class. Instance variable is passed on init.

Nothing in Python makes it possible to enforce data hiding. It has no data privacy, it only has data obfuscation.

Any value in python is an object, and it has a class (which is also called its type). It is stored as object\_name.\_\_class\_\_. E.g.- if we do i=5, doing i.\_\_class\_\_ gives “class-int”.

There are two builtin functions in python which work on the concept of inheritance.

isinstance() and issubclass()

issubclass(bool,int) is True, bool is subclass of int.

isinstance(object\_name, class\_name)

How does Python decide on the scope of the variables:

Python doesn't have variable declarations, so it has to figure out the scope of variables itself. It does so by a simple rule: If there is an assignment to a variable inside a function, that variable is considered local. Say you define x outside the function and x is a global variable, and if you again do x=x+1, it gives an error, because as soon as you start assigning a value to x inside the function definition, x is treated on local scope. But if you just print x, then the global value is used. For assignment it would not work, but you can use it.

\_\_fn\_\_ = for this fn, user manipulation is not possible. Non-callable functions, only pythons works on those functions. Hidden function.

To unhide the \_\_attributes of a class-

print(obj.\_classname\_\_attribute name)

Attribute is either a class variable or a class function.

Difference between class variable and instance variable is that , class variables are static data members and remain same for all the objects. Instance variables change with each instantiation of an object.

Class variables are shared by all the instances, so we should not use nonmutable data types as class variables.

If using nonmutable types, use it for instance variable – do self.variable = []. If you use self, for each instance, that is perfectly fine.

Doing self.variable = variable , creates the variable inside the instance of the class. If we don't do that , variable is just created inside the function.

Data attributes will be replaced by method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

## Magic methods of class:

The str() returns representations of values which are fairly human-readable, while repr() is meant to (or will force a SyntaxError if there is no equivalent syntax). Many values, such as numbers or structures like

\_\_str\_\_ human-readable

\_\_repr\_\_ helps to recreate the object, generate representations which can be read by the interpreter  
To change the string representation of an instance, define the \_\_str\_\_() and \_\_repr\_\_() methods. We have a string casting for our classes and we can simply print out instances. lists and dictionaries, have the same representation using either function. Strings, have two distinct representations. repr adds string quotes and backslashes.

str(p1) or format(p1), Python is internally doing p1.\_\_str\_\_().

obj\_name.\_\_class\_\_.\_\_name\_\_

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. Method definitions will bind the name of the methods here.

A class object is a wrapper around the contents of this namespace created by the class definition, the original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound to the class name.

Objects – Instantiation, attribute references

Instantiation :Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. x = MyClass()

This creates an empty object. Many classes like to create objects with instances customized to a specific initial state. So use \_\_init\_\_

attribute references: You can access the attributes of the class using object

Class\_name.fn\_name will give the function object, as defined in the class. You can store this fn object and call it later. i.e x\_fo=x.f.   
Call using – x\_fo()

To call that fn, Class\_name.fn\_name().

Attributes = variables+functions.

Variables = data attributes, data members, instance variables.  
Functions = methods

A class is a collection of objects of a similar type. Once a class is defined, any number of objects can be created which belong to that class. A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

The instance is the actual object created at runtime. One can have an instance of a class or a particular object. More than one instance of the same class can be in existence at any one time. Instance: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Objects are the basic run-time entities. An object is an instance of a class. A class must be instantiated into an object before it can be used in the software.

Objects are "black boxes." The underlying implementations of objects are hidden from the users of the object and is only visible to the creators of the object.

Method describes the object’s abilities.

Operator overloading – the operator works on the objects. Create two objects of the same class and use the overloaded operator on them to see the results.

List is also a class, and its methods are append, insert etc.. Similarly for other datatypes.

classes also use dynamic nature of Python: they are created at runtime, and can be modified further after creation.

class members – public

member functions – virtual

Arguments in the class:

1. Arguments passed to the \_\_init\_\_ method of class.  
   these can be used by all the methods of the class directly by self.arg\_name  
   But while instantiating object, make sure to pass them to the class obj\_name=class\_name(args)
2. Arguments passed to individual methods  
   only used by those individual methods
3. Self is to be passed to all the methods of the class.

Methods in Class:

1. User defined methods   
   def user\_defined\_name
2. Built in methods  
   builtin names , \_\_builtin\_name\_\_

Common thing for both the methods is – self is passed to both.

Why to pass self – while you instantiate an object, obj\_name=class\_name() , what does “()” mean, it means, you are passing self(the obj\_name itself) to the class. But inside the (), you do not write self (or obj\_name) again, because python does it for you.  
we need not do obj\_name=class\_name(obj\_name).

Classes themselves are objects. This provides semantics for importing and renaming. Unlike C++, built-in types can be used as base classes which can be inherited by the user.

Aliasing is present in python. An alias is a second name for a piece of data, which is easier than creating a copy for the same data. If the data is immutable—i.e., if it cannot be modified in place—then aliasing doesn't matter.

aliasing happens whenever one variable's value is assigned to another variable

first = [1,2,3]

second = first

changes in first, will also effect second.

A namespace is a mapping from names to objects.

z.real - real is the attribute of z.  
Function in a module is an attribute of the module object as it is referenced by modname.fnname.

Namespaces are created at different moments and have different lifetimes.

Eg - built-in namespace 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.

Although scopes are defined statically, they are accessed dynamically. Order of scope search:

1. innermost scope
2. non-local
3. next-to-last scope contains the current module’s global names.(Global)
4. built-in scope

LEGB Local Enclosed Global Builtin

Remember that Assignments do not copy data — they just bind names to objects. The global statement can be used to indicate that particular variables live in the global scope and should be rebound there.

Depth-first, Left-to-Right is the MRO in classes. A(B1,B2..). A first then parents of A, next B1, parents of B1, next B2, parents of B2 and so on.

\_spam – internal use, weak "internal use" indicator. It should be considered an implementation detail and subject to change without notice.

Double underscore – Name mangling. Cannot be accessed with just \_\_variable\_name, access it using object. \_className\_\_variableName

\_\_x\_\_ = Builtin functions, magic methods

To the various objects of a class, you can add a member variable to that object by doing:

A is the class, a=A() # a is the object

a.\_\_dict\_\_[‘name’] = ‘foo’

**MetaClass:**

Instance = instance of class

Class = instance of metaclass

Class is also an object. This class object is an instance of metaclass.

E.g.- List is a class, type is a metaclass of list.

Magic methods - allow the programmer to override behaviour for various operators and behaviour of objects.

A scope is a region in a program , where a namespace is directly accessible. Scope are defined statically, but are used dynamically.

If you del x, implies that the binding of x from the namespace referenced by the local scope is removed(just the binding is removed, not the object).

>>> x=42>>> id(x) is 1883567184>>> del x>>> y=42>>> id(y) is same 1883567184

The method of a class should have self as the first argument representing the object, when you call this method, the self is provided implicitly by the call.

Functions defined in the class, do not become methods directly.   
Function object + instance object = method object.

Be careful about the difference between class variable and instance variable, because class variables can be modified by all the objects of the class, whereas instance variable can be modified only by that object defined by that instance.

Class variable – simple variable\_name, instance variable is self.variable\_name.

Importance of \_\_init\_\_ = if you do object\_name=Class\_name(), this creates an empty object, but if you add an init method, that object will have certain initial state as defined in the init method.

Self cannot be accessed outside the methods of the class. self. can only be done inside the class’s methods.

Singleton class is a class where only one instance creation is possible. You can implement such class using a decorator singleton which takes in a class and returns a function.  
def singleton(cls):  
 instances = {}  
 def getinstance():  
 if cls not in instances:  
 instances[cls] = cls()  
 return instances[cls]  
 return getinstance

If we do @singleton before the class definition(say class Abc), if we do c=Abc(), and d = Abc(). If we give c==d , it gives True because both are same.

Applications of singleton - We use a singleton when we need to manage a shared resource. Eg- printer spooler. There should only be a single instance of the spooler in order to avoid conflicting requests for the same resource. Or a database connection or a file manager etc for single point of communication.

Class has some special attributes, like

\_\_bases\_\_ = tells about the base classes from which the class inherits

\_\_dict\_\_ = dictionary for class’s namespace

\_\_name\_\_ = string of class’s name

\_\_module\_\_ = file name in which the class is defined

\_\_class\_\_ = Class from which the object is instantiated.

If you do want to give direct access to a particular class variable, make it self.\_attributename = value

To know how many objects of a particular class are created, take a counter variable and increment it in the \_\_init\_\_ method. That counter variable should be class attribute and in the \_\_init\_\_ method, do – className.classattributeName+=1

We can validate the variables passed to the class, in the \_\_init\_\_ method definition. if x<10, self.x=x else, raise ValueError. This can be done.

To inherit from the basic datatypes – in the init method of the base class , do list.\_\_init\_\_(self) (if you are inheriting from list type).

If you are passing value\_name to the \_\_init\_\_ method. You can separate value\_name as self.v=value\_name[:4] and self.n=value\_name[5:]. So one single variable(value\_name) can be split to two different variables of self.

# TELNETLIB:

Do not reopen an already connected instance.

The '\r' character is the carriage return, and the carriage return-newline pair is both needed for newline in a network virtual terminal session.

tn.write(rnc\_password + '\r\n')

tn.read\_until(">",timer)

Logging – developer says that if so and so event happens, log that information to a stream . that event’s importance is decided by the developer.

Debug,info, warning, error, critical(DIWEC)

If u set to the lower level, all the higher order messages will be printed out. If level is warning, only error and critical messages are printed out.

class PrimesTestCase(unittest.TestCase):

self.assertTrue(is\_prime(5))

# Linux Commands:

Ps – shows information about the currently running processes.

ps [options]

-a option tells ps to list the processes of all users on the system

-u option tells ps to provide detailed information about each process.

-x option adds to the list processes that have no controlling terminal, such as daemons, which are programs that are launched during booting (i.e., computer startup) and run unobtrusively in the background until they are activated by a particular event or condition.

the output of ps -aux can be piped (i.e., transferred) to the less command,

ps -aux | less

information that ps -aux provides about each process

ps -eaf | grep OutageLogCollection

oamops 17024 15917 0 12:02:15 pts/28 0:00 grep OutageLogCollection

user/ percentage of CPU used by the process, the percentage of memory used by the process, VSZ (virtual size in kilobytes), RSS (real memory size or resident set size in 1024 byte units), STAT (the process state code), the starting time of the process, the length of time the process has been active and the command that initiated the process. The process state codes include D, uninterruptable sleep; N, low priority; R, runnable (on run queue); S, sleeping; T, traced or stopped; Z, defunct (zombie).

# Threading:

Using threads allows a program to run multiple operations concurrently in the same process space. The simplest way to use a Thread is to instantiate it with a target function and call start() to let the thread begin working.

t = threading.Thread(target=worker)

t.start()

Usually our main program implicitly waits until all other threads have completed their work. However, sometimes programs spawn a thread as a daemon that runs without blocking the main program from exiting. Using daemon threads is useful for services where there may not be an easy way to interrupt the thread or where letting the thread die in the middle of its work does not lose or corrupt data (for example, a thread that generates “heart beats” for a service monitoring tool). To mark a thread as a daemon, call its setDaemon() method with a boolean argument. The default is for threads to not be daemons, so passing True turns the daemon mode on. By setting them as daemon threads, we can let them run and forget about them, and when our program quits, any daemon threads are killed automatically. The significance of this flag is that the entire Python program can exit even if daemon threads are left.

def \_\_init\_\_(self, group=None, target=None, name=None,

args=(), kwargs=None, \*, daemon=None):

t = threading.Thread(target=f, args=(i,))   
t.start()

A thread object can be started only once. t.start() can be used only once on a thread object t.

args is (i,) because args is a tuple, and those args can be used to loop through the thread. Those args are passed to the function, not to the thread, so define the target function that it accepts arguments.

t.start() is the basic way to start a thread.

The ''t.join()'' method blocks the main application execution until the thread t exits. t is a ‘thread object’.

-->start() method starts a completely new thread and calls run() method to execute it, hence obeys the basic law of threading.

--> In case of run() method, no new thread is created and it executes on the current thread.

multiple threads accessing the same resource will contend for that resource. Contention leads to deadlocks.

## GIL:

Global Interpreter Lock or GIL, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter to prevent deadlock. This means that only one thread can be in a state of execution at any point in time.  
A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution.

If you want to run some things simultaneously, and efficiency is not a concern, then use threading module. But the threading library won’t let you use extra CPU cores.  
To avoid GIL problem use multiprocessing, each Python process gets its own Python interpreter and memory space so the GIL won’t be a problem. Multiple processes are heavier than multiple threads because of overheads.

The threading.Thread class has the following methods-

1. run Entry point for thread
2. start start calls run to start the thread
3. join join waits for threads to terminate
4. isAlive if a thread is still executing
5. getName gives the name of the thread
6. setName sets the name of the thread

# Subprocess:

Stdin, stderr, stdout are known as pipes

How to redirect output from stdout, stderr to files.

Has a class Popen

class Popen(args, bufsize=-1, executable=None,

stdin=None, stdout=None, stderr=None,

preexec\_fn=None, close\_fds=True, shell=False,

cwd=None, env=None, universal\_newlines=False,

startupinfo=None, creationflags=0,

restore\_signals=True, start\_new\_session=False, pass\_fds=())

# TIME:

Time.gmtime(0)

from collections import Counter

c=Counter('abcabcdfret')

sorted(c) will give element by element without repetition

c[‘a’] will give the count of a

Counters are similar to dictionary, except that they don’t give traceback error if you access a key which is not present in the counter.

1. A Counter is a dict subclass for counting hashable objects. present in module collections  
2. you can define a counter object by C=Counter()  
3. You can add and subtract the counter elements, while subtracting be careful to see which count is bigger  
4. You can convert a counter object to dictionary object using dict()

# Concurrency:

What is concurrency? Concurrency, when applied to application/program logic, is the simultaneous execution of tasks. For the most part, these tasks interact and pass information to and from each other and the parent. Parallelism, when applied to applications, generally refers to the actual implementation of concurrent programming. Specifically, it refers to the simultaneous execution of tasks in such a way as to take advantage of multiple processors, multiple processor cores, or even multiple machines within a computing grid.

Threads over processes, threads share memory and other resources. A single process can have multiple threads. If opening a word doc is a process, one thread runs in the foreground to accept user input, and the other thread runs in the back.

Pthread is a worker which is spawned by the parent process, and which shares that parent's resources.

Threads run by the interpreter = green threads  
by the OS = native threads

Threads use data sharing; everyone has everything. Threads, just like processes, are designed to be passed off to the operating system scheduler, which dictates which CPU a given process is run on, when it is run on that CPU which means that not only do you gain shared memory with threads, you can leverage all of the local machine's resources (in theory) including multiple cores and/or CPUs.

# Time Complexity:

1. Finding length of a list is O(1) – as length is maintained as an attribute and each time a new item is added, the counter is increased.
2. Timsort – O(nlogn)
3. If you already know the length of the output list beforehand, define a list as [0]\*n and then set the items as and when needed. O(1) for set operation and O(n) for reverse.
4. Always reduce the input to be tested by half by using binary search technique. Whenever comparison over a range in involved, divide the range by 2 using binary search.

# Miscellaneous:

input() is always string, even if you enter an int

a, b, c = 1, 2, "john"   
Multiple variables can be initialised. No need to use semicolon in python. Tuple packing unpacking concept.

If a = 1, 2, "john", then a is a tuple – this is just packing.

def fib(n): # write Fibonacci series up to n  
a, b = 0, 1  
while a < n:  
print a  
a, b = b, a+b  
# Now call the function we just defined:  
fib(2000)

def reversed\_string(a\_string): return a\_string[::-1]

str.rjust(50, '0') == right justifies by 50 places

Palindrome : if(str\_name == str\_name[::-1])

OR - word\_rev = reversed(word)

word should be a sequence, *reversed* is a keyword which generates reversed iterator object.

For case insensitive situations - string.upper() == string.upper()[::-1]

FOR loop in strings:

ix = 0  
while ix < len(fruit):  
 letter = fruit[ix]  
 print(letter)  
 ix += 1

For loop in lists:

combs = []

for x in [1,2,3]:

for y in [3,1,4]:

if x != y:

combs.append((x, y))

Sum function call in a recursive way-   
return args[-1] + mySum(\*args[:-1])

Loop in dictionary:

for name, number in phonebook.iteritems():

print "Phone number of %s is %d" % (name, number)

to make a list of a particular range – list(range(x,y))

s=lambda x: x > 10 and x < 80

This lambda, defines a fn as –x is a variable to be passed to the fn ((lambda x:) and it follows a rule (x>10 and x<80).

Lambda function can take multiple arguments as input, just like that of functions, only difference being – you have to wirte the logic in one line.

s(81) gives False. s(11) gives True.

for line in open("myfile.txt"):

print(line, end='')

end is needed because python by default adds \n, apart from the newline present in the file, extra newline is added, which makes it 2newlines, so if we add end, instead of adding default \n python will add nothing, which prints the content in the file as is.

## Vi commands

w=Next word

b=Beginning of word

e=End of word

{=Move to a paragraph forward

}=Move to a paragraph backward

G=move to the last line

1G=Move to the first line

fc=Move to the character c

%=Move to corresponding braces

x=Delete to right of cursor

X=Delete to left of cursor

D=Delete everything till the end of line

dd=delete current line

/=forward search

?=backward search

n=next search

N=previous search

## Stubbing:

Stubbing is defining a function, but not writing the logic in it, just do – pass. It is useful in software development, to provide skeleton for the program.

## Pickling:

Import pickle

2 methods – load, dump (obj to file)

Pickle.dump(obj\_name.file\_object)

Obj2\_name = pickle.load(file\_object)

Obj2 is same as obj

Python obj to string – string can be used as a dictionary key.

## Descriptors:

We have an object named object1, it has an attribute attribute1. So a descriptor holds the value of object1.atttribute1.

Descriptor is a class with \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_ methods.

Objects having these methods = their meaning

\_\_get\_\_ = Only read

\_\_get\_\_ and \_\_set\_\_ = Writable

descriptor is a way to customize what happens when you reference an attribute on a model. You might need to validate the attribute value. You may want to retrieve the attribute value and cache it for later use, so that future references don’t have all the overhead.

\_\_get\_\_(self, instance, owner) — This will be called when the attribute is retrieved (value = obj.attr), and whatever it returns is what will be given to the code that requested the attribute’s value. Owner is owner class.

\_\_set\_\_(self, instance, value) — This gets called when a value is set to the attribute (obj.attr = 'value'), and shouldn’t return anything at all.

\_\_delete\_\_(self, instance) — This is called when the attribute is deleted from an object (del obj.attr)

getattr:

list1=[1,2,3]

getattr(list1, "append")(4)

list1 is now equal to [1,2,3,4]

getattr(list1, "append") returns the method ‘append’ and you can pass arguments to this method.

Remember that attributes of an object could be a method or a variable.

getattr(object\_name, attribute\_name, default\_value)

#significance of default\_value = if the object doesn’t have the given attribute, return this default value.

setattr(object\_name, attribute\_name, value\_tobe\_set)

Remember that these are used to get and set the instance attributes, not the class attributes.

property is a built-in function

name = property(get\_name, set\_name, delete\_name, "Publisher name")

get\_name, set\_name, delete\_name are the functions we define in the class to get, set and delete the attributes respectively.

As soon as you give this line to the ‘name’ variable – it says that- whenever you want to fetch the value of obj.name go to get\_name function and whenever you want to set the value of name (obj.name=something) go to set\_name.

Accessors and mutators are often called getters and setters.

mysqldb

44.How to retrieve data from a table in MySQL database through Python code? Explain.

1. import MySQLdb module as : import MySQLdb

2. establish a connection to the database.

db = MySQLdb.connect(“host”=”local host”, “database-user”=”user-name”, “password”=”password”, “database-name”=”database”)

3. initialize the cursor variable upon the established connection: c1 = db.cursor()

4. retrieve the information by defining a required query string. s = “Select \* from dept”

5. fetch the data using fetch() methods and print it. data = c1.fetch(s)

6. close the database connection. db.close()

## Theory:

1. Zen of python – has the doctrine of python

import this

#will give the doctrine statements

1. Sequence unpacking has the limitation – you have to give the exact number of variables to be unpacked. If the variables have to be unpacked without specifying the exact numbers.

Do a, \*b = “string1”.split()

\*b accepts variable number of arguments. \*b has to be placed at the end.

1. Shell is nothing but a set of programs that executes system commands on user’s behalf

# Practical Points:

1. Never use module names (builtin) as the name of the file. Because when you do import, it will import the file, instead of the builtin.  
   As Scope is LEGB – Local Enclosed Global builtin
2. Why do we declare variables – to know how much memory is to be assigned to each type of variable, based on the variable, we would know how much memory is to be assigned to each type

# References:

* 1. https://www.cs.toronto.edu/~david/course-notes/csc110-111/05-memory-model/03-python-memory-model-1.html