COURSE CURRICULUM

1. Basics of Python

Motivation & Applicability to various domains, Installation & Setting up path Input / Output

Python is a highly versatile programming language with many use cases. It's important to have a specific **motivation** in mind as you embark on your **Python** learning journey. Targeting an area of interest will help you get started quicker and filter out learning resources that don't fit your use case.

Python is used in many application **domains**. Here's a sampling. The **Python** Package Index lists thousands of third party modules for **Python**. Web and Internet Development. **Python** offers many choices for web development: Frameworks such as Django and Pyramid. Micro-frameworks such as Flask and Bottle. Advanced content management systems such as Plone and django CMS.

Python source code is available under the GNU General Public License (GPL). Python interpreter is free, and downloads are available for all major platforms (Windows, Mac OS, and Linux) in the form of *source* and *binary*. You can download it from the Python Website: python.org.

Before starting working with Python, a specific path is to set.

- Your Python program and executable code can reside in any directory of your system, therefore Operating System provides a specific search path that index the directories Operating System should search for executable code.
- The Path is set in the Environment Variable of My Computer properties:
- To set path follow the steps:

Right click on My Computer ->Properties ->Advanced System setting ->Environment Variable ->New

In Variable name write path and in Variable value copy path up to C://Python(i.e., path where Python is installed). Click Ok ->Ok.

Path will be set for executing Python programs.

1. Right click on My Computer and click on properties.

2. Click on Advanced System settings

3. Click on Environment Variable tab.

4. Click on new tab of user variables.

5. Write path in variable name

6. Copy the path of Python folder

7. Paste path of Python in variable value.

8. Click on Ok button:

2. Keywords and Identifiers, Variables and Data Types

Python has a set of keywords that are reserved words that cannot be used as variable names, function

Names, or any other identifiers. Keywords are case sensitive in python.Here is the list of keywords in python:

Keywords in Python				
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	

False:

The False keyword is a Boolean value, and result of a comparison operation.

The False keyword is the same as 0 (True is the same as 1).

Ex: print(5 > 6)

O/p : False

None:

The Nonekeyword is used to define a null value, or no value at all.

None is not the same as 0, False, or an empty string. None is a data type of its own (NoneType) and only None can be None.

Ex: x = None

print(x)

O/p: None

True:

The True keyword is a Boolean value, and result of a comparison operation.

The True keyword is the same as 1 (False is the same as 0).

Ex: print(7 > 6)

o/p: True

And:

The and keyword is a logical operator.

Logical operators are used to combine conditional statements.

The return value will only be True if both statements return True, otherwise it will return False.

Ex: x = (5 > 3 and 5 < 10)

print(x)

o/p: True

Or:

The or keyword is a logical operator.

Logical operators are used to combine conditional statements.

The return value will be True if one of the statements return True, otherwise it will return False.

Ex: x = (5 > 3 or 5 > 10)

print(x)

o/p: True

As:

The as keyword is used to create an alias.

In the example above, we create an alias, c, when importing the calendar module, and now we can refer

to the calendar module by using c instead of calendar.

Ex:

import calendar as c

print(c.month_name[1])

o/p: January

Assert:

The assert keyword is used when debugging code.

The assert keyword lets you test if a condition in your code returns True, if not, the program will raise

an AssertionError.

Ex: x = "hello"

#if condition returns False, AssertionError is raised:

assert x == "goodbye", "x should be 'hello'"

o/p:Traceback(most recent call last):

File “demo_ref_keyword_assert2.py”,line 4,in<module>

assertx=”goodbye”,”x should be ‘hello”

AssertionError:x should be hello’

Break:

The break keyword is used to break out afor loop, or a while loop.

Ex: i = 1

while i< 9:

print(i)

if i == 3:

break

i += 1

o/p:1

2

3

Class:

The class keyword is used to create a class.

A class is like an object constructor.

Ex: class Person:

name = "John"

age = 36

print(Person.name)

o/p: John

Continue:

The continue keyword is used to end the current iteration in a for loop (or a while loop), and continues to the next iteration.

Ex:for i in range(9):

if i == 3:

continue

print(i)

o/p:0

1

2

3

4

5

6

7

8

Def:

The def keyword is used to create, (or define) a function.

Ex:def my_function():

print("Hello from a function")

my_function()

o/p: Hello from a function

Del:

The del keyword is used to delete objects. In Python everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list etc.

Ex: class MyClass:

```
    name = "John"
    del MyClass
print(MyClass)
```

o/p: name error:name ‘MyClass’ is not defined

Elif:

The elif keyword is used in conditional statements (if statements), and is short for else if.

```
Ex: for i in range(-5, 5):
  if i> 0:
    print("YES")
elifi == 0:
    print("WHATEVER")
  else:
    print("NO")
```

o/p:NO

```
      NO
    NO
    NO
    NO
   WHATEVER
   YES
   YES
   YES
   YES
```

Else:

The else keyword is used in conditional statements (if statements), and decides what to do if the

condition is False.

Ex:x = 2

if x > 3:

  print("YES")

else:

  print("NO")

o/p:NO

Finally:

The finally keyword is used in try...except blocks. It defines a block of code to run when the

try...except...else block is final.

The finally block will be executed no matter if the try block raises an error or not.

Ex:try:

  x > 3

except:

print("Something went wrong")

else:

print("Nothing went wrong")

finally:

print("The try...except block is finished")

o/p:

Something went wrong

The try.....except block is finished

For:

The for keyword is used to create a for loop.

It can be used to iterate through a sequence, like a list, tuple, etc.

Ex:for x in range(1, 9):

print(x)

o/p:1

2

3

4

5

6

7

8

From:

The from keyword is used to import only a specified section from a module.

Ex:from datetime import time

x = time(hour=15)

print(x)

o/p:15:00:00

Global:

The global keyword is used to create global variables from a no-global scope, e.g. inside a function.

Ex:#create a function:

def myfunction():

```
  global x

  x = "hello"

#execute the function:

myfunction()

#x should now be global, and accessible in the global scope.

print(x)
```

o/p:Hello

If:

The if keyword is used to create conditional statements (if statements), and allows you to execute a block of code only if a condition is True.

```
Ex:x = 5

if x > 3:

  print("YES")
```

o/p:YES

Import:

The import keyword is used to import modules.

```
Ex:import datetime

x = datetime.datetime.now()

print(x)
```

o/p:

In:

The in keyword has two purposes:

The in keyword is used to check if a value is present in a sequence (list, range, string etc.).

Ex: fruits = ["apple", "banana", "cherry"]

if "banana" in fruits:

print("yes")

o/p: Yes

Is:

The is keyword is used to test if two variables refer to the same object.

The test returns True if the two objects are the same object.

The test returns False if they are not the same object, even if the two objects are 100% equal.

Use the == operator to test if two variables are equal.

Ex:x = ["apple", "banana", "cherry"]

y = x

print(x is y)

o/p: True

Lambda:

The lambda keyword is used to create small anonymous functions.

A lambda function can take any number of arguments, but can only have one expression.

The expression is evaluated and the result is returned.

Ex:x = lambda a : a + 10

print(x(5))

o/p:15

Nonlocal:

The nonlocal keyword is used to work with variables inside nested functions, where the variable

Should not belong to the inner function.

Use the keyword nonlocal to declare that the variable is not local.

Ex: def myfunc1():

x = "John"

def myfunc2():

 nonlocal x

x = "hello"

myfunc2()

return x

print(myfunc1())

o/p:hello

Not:

The not keyword is a logical operator.

The return value will be True if the statement(s) are not True, otherwise it will return False.

Ex: x = False

print(not x)

Or:

The or keyword is a logical operator.

Logical operators are used to combine conditional statements.

The return value will be True if one of the statements return True, otherwise it will return False.

Ex:x = (5 > 3 or 5 > 10)

Print(x)

o/p: True

Pass:

The pass statement is used as a placeholder for future code.

When the pass statement is executed, nothing happens, but you avoid getting an error when empty

code is not allowed.

Empty code is not allowed in loops, function definitions, class definitions, or in if statements.

Ex:for x in [0, 1, 2]:

    pass

having an empty for loop like this, would raise an error without the pass statement

Raise:

The raise keyword is used to raise an exception.

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

Ex:x = -1

if x < 0:

 raise Exception("Sorry, no numbers below zero")

o/p: Exception- sorry no numbers below zero

Return:

The return keyword is to exit a function and return a value.

Ex: def myfunction():

return 3+3

print(myfunction())

o/p:6

Return:

The return keyword is to exit a function and return a value.

Ex:def myfunction():

    return 3+3

print(myfunction())

o/p:

Try:

The try keyword is used in try...except blocks. It defines a block of code test if it contains any errors.

Ex: # (x > 3) will raise an error because x is not defined:

    try:

    x > 3

    except:

print("Something went wrong")

print("Even if it raised an error, the program keeps running")

o/p: Something went wrong

     Even if it raised an error, the program keeps running

With:

Used to simplify exception handling

While:

The while keyword is used to create a while loop.

A while loop will continue until the statement is false.

Ex: x = 0

```
while x < 9:

 print(x)

 x = x + 1
```

Yield:

To end a function, returns a generator

Identifiers:

All the variables, class, object, functions, lists, dictionaries etc. in Python are together termed as Python Identifiers. Identifiers are the basis of any Python program. Almost every Python Code uses some or other identifiers.

Rules for Indentifiers:

- An identifier name should not be a keyword.
- An identifier name can begin with a letter or an underscore only.
- An identifier name can contain both numbers and letters along with underscores (A-z, 0-9, and _).
- An identifier name in Python is case-sensitive i.e, sum and Sum are two different identifier.

Ex: a = “hello world”

```
 Print a
```

o/p: hello world

Variables:

Variables are containers for storing data values.

Syntax:x = 5

y = "John"

print(x)

print(y)

o/p: 5

```
 john
```

Data Types:

Variables can store data of different types, and different types can do

different things.

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Conditional Statements:

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

if

An "if statement" is written by using the if keyword.

Syntax: a = 33

b = 200

if b > a:

print("b is greater than a")

o/p: b is greater than a

elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Ex: a = 33

b = 33

if b > a:

print("b is greater than a")

elif a == b:

print("a and b are equal")

o/p: a and b are equal

Else

The else keyword catches anything which isn't caught by the preceding conditions.

Ex:a = 200

b = 33

if b > a:

print("b is greater than a")

elif a == b:

print("a and b are equal")

else:

print("a is greater than b")

o/p: a is greater than b

And

The and keyword is a logical operator, and is used to combine conditional statements:

Ex: a = 200

b = 33

c = 500

if a > b and c > a:

print("Both conditions are True")

o/p: Both conditions are true

or

The or keyword is a logical operator, and is used to combine conditional statements:

Ex: a = 200

b = 33

c = 500

if a > b or a > c:

print("At least one of the conditions is True")

o/p: Atleast one of the condition is true

Nestedif

You can have if statements inside if statements, this is called nested if statements.

Ex: x = 41

if x > 10:

print("Above ten,")

if x > 20:

print("and also above 20!")

else:

print("but not above 20.")

o/p: above 10

and also above 20

Pass

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
Ex: a = 33
    b = 200
if b > a:
   pass
# having an empty if statement like this, would raise an error without the pass statement
```

Looping

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

```
Ex: i = 1
while i< 6:
 print(i)
i += 1
```

o/p: 1

2

3

4

5

Note: remember to increment i, or else the loop will continue forever.

Break Statement

With the break statement we can stop the loop even if the while condition is true

Continue Statement

With the continue statement we can stop the current iteration, and continue with the next

For Loop:

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Ex: fruits = ["apple", "banana", "cherry"]

for x in fruits:

print(x)

o/p: apple

banana

cherry

The break statement

With the break statement we can stop the loop before it has looped through all the items

Continue statement

With the continue statement we can stop the current iteration of the loop, and continue with the next

Strings and features

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Ex: print("Hello")

print('Hello')

o/p: Hello

Hello

String methods:

Python has a set of built-in methods that you can use on strings.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where

	it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable

isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it found
rindex()	Searches the string for a specified value and returns the last position of

	where it found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string

upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

String Manipulation, Functions

If you have been exposed to another programming language before, you might have learned that you need to declare or *type* variables before you can store anything in them. This is not necessary when working with strings in Python. We can create a string simply by putting content wrapped with quotation marks into it with an equal sign (=):

Message = “hello world”

Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Ex: def my_function():
  print("Hello from a function")

2)Python collections and sequences: -

1) Lists and features:

List:

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.). A list can also have another list as an item. This is called a nested list.

thislist = ["apple", "banana", "cherry"]
print(thislist)

output: [‘apple’, ‘banana’, ‘cherry’]

append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Features:

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

2) List Functions:

- The list() function creates a list object.
- A list object is a collection which is ordered and changeable.

Syntax

list(iterable)

Note: iterable Required. A sequence, collection or an iterator object

3) List Comprehension:

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example: Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)
```

output:
['apple', 'banana', 'mango']

The syntax:

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

4) Tuple and Features:

Tuple:

Tuples are used to store multiple items in a single variable.

- Tuple is one of 4 built-in data types in Python used to store collections of data, the are list, set and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and unchangeable**.**
- Tuples are written with round brackets.

Example:

thistuple = ("apple", "banana", "cherry")

print(thistuple)
output: ('apple', 'banana', 'cherry')

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Features:

- Iterating through a tuple is faster than with a list since tuples are immutable.

- Tuples that consist of immutable elements can be used as a key for the dictionary.
- If you have data that is immutable, implementing it as a tuple will guarantee that it remains write-protected

5) Functions and Methods:

Functions:

In Python, a function is a group of related statements that performs a specific task.

- With the help of Functions, we can break our program into smaller and modular chunks. So, when our program grows larger and larger, functions help us to make it more organized and manageable.
- Furthermore, with the help of function, we can avoid repetition and makes our code reusable.
- Mainly, there are two types of functions:
- User-defined functions – These functions are defined by the user to perform a specific task.
- Built-in functions – These functions are pre-defined functions in Python.

Creating a function:

To define the function in Python, it provides the def keyword. Now, let's see the syntax of the defined function.

Syntax:

def my_function(parameters):

function_block

return expression

Methods:

- Method is called by its name, but it is associated to an object (dependent).
- A method is implicitly passed the object on which it is invoked.
- It may or may not return any data.
- A method can operate on the data (instance variables) that is contained by the corresponding class.

Basic Method Structure in Python:

class class_name

```
def method_name () :
                    ......
                    # method body
                        ......
```

6) Dictionaries:

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

Example:

Create and print a dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

7) Sets and Frozen sets:

Sets:

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is both *unordered* and *unindexed*.
- Sets are written with curly brackets.

Example:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

output:
{'banana', 'apple', 'cherry'}

Different methods in sets:

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not

issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Frozen set:
The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable).

Syntax:

frozenset(*iterable*)

Example:

Try to change the value of a frozenset item.

p

```
mylist = ['apple', 'banana', 'cherry']
x = frozenset(mylist)
x[1] = "strawberry"
```

output:

```
Traceback (most recent call last):
  File "demo_ref_frozenset2.py", line 3, in <module>
    x[1] = "strawberry"
TypeError: 'frozenset' object does not support item assignment
```

3)Working with python collections

1)Working with lists and tuples

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3

are Tuple, Set, and Dictionary, all with different qualities and usage.

```
Ex:thislist = ["apple", "banana", "cherry"]
 print(thislist)
```

o/p:['apple','banana','cherry']

Access List items

List items are indexed and you can access them by referring to the index number:

```
Ex:thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

o/p: banana

Change List items

To change the value of a specific item, refer to the index number:

```
Ex:thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

o/p: ['apple','blackcurrent','cherry']

Add List items

To add an item to the end of the list, use the append() method:

Ex:thislist = ["apple", "banana", "cherry"]

thislist.append("orange")

print(thislist)

o/p:['apple','banana','cherry','orange']

Remove List items

The remove() method removes the specified item.

Ex:thislist = ["apple", "banana", "cherry"]

thislist.remove("banana")

print(thislist)

o/p: ['apple','cherry']

Loop Lists

You can loop through the list items by using a for loop:

Ex: thislist = ["apple", "banana", "cherry"]

for x in thislist:

  print(x)

o/p:apple

banana

cherry

Sort Lists

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

Ex:thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort()

print(thislist)

o/p: ['banana','kiwi','mango','orange','pineapple']

Python has a set of built-in methods that you can use on lists.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list

sort()	Sorts the list

Tuples

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

Ex: thistuple = ("apple", "banana", "cherry")

print(thistuple)

o/p: ('apple','banana','cherry')

Tuple methods:

Python has two built-in methods that you can use on tuples.

Method	**Description**
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it found

2)Working with Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

Ex: thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(thisdict)

o/p:{'brand' : 'ford', 'model':'mastang',' year':1964}

Dictionary methods:

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys

pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Working with sets and frozen sets

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is both *unordered* and *unindexed*.

Sets are written with curly brackets.

Ex:thisset = {"apple", "banana", "cherry"}

print(thisset)

Note: the set list is unordered, meaning: the items will appear in a random order.

Refresh this page to see the change in the result.

o/p:{'apple','banana','cherry'}

Set methods:

Python has a set of built-in methods that you can use on sets.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more s
difference_update()	Removes the items in this set that are also included in another
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, spe
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not

pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Frozenset

The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable).

Ex: mylist = ['apple', 'banana', 'cherry']

x = frozenset(mylist)

print(x)

o/p:frozenset({'apple','banana','cherry'})

4)Python functional programming:

1)Types of functions:

There are three types of functions in Python:

- Built-in functions, such as help() to ask for help, min() to get the minimum value, print() to print an object to the terminal,... You can find an overview with more of these functions here.
- User-Defined Functions (UDFs), which are functions thatusers create to help them out; And
- Anonymous functions, which are also called lambda functions because they are not declared with the standard def keyword.

2) Function Arguments:

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

output:

Emil Refsnes

Tobias Refsnes

Linus Refsnes

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example:

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)


my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)


my_function("Emil")
```

Traceback (most recent call last):

File "demo_function_args_error.py", line 4, in <module>

```
 my_function("Emil")
```

TypeError: my_function() missing 1 required positional argument: 'lname'

3)Anonymous functions:

In Python, an anonymous function means that a function is without a name. As we already know that the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in Python.

It has the following syntax:

Syntax:
lambda arguments: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Example:

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y

# using the normally
# defined function
print(cube(5))

# using the lambda function
print(lambda_cube(5))
```

4) Special functions (map, reduce, filter):

Map, Filter, and Reduce are paradigms of functional programming. They allow the programmer (you) to write simpler, shorter code, without neccessarily needing to bother about intricacies like loops and branching.

Essentially, these three functions allow you to apply a function across a number of iterables, in one fell swoop. map and filter come built-in with Python (in the __builtins__ module) and require no importing. reduce, however, needs to be imported as it resides in the functools module. Let's get a better understanding of how they all work, starting with map.

Map

The map() function in python has the following syntax:

map(func, *iterables)

Where func is the function on which each element in iterables (as many as they are) would be applied on. Notice the asterisk(*) on iterables? It means there can be as many iterables as

possible, in so far func has that exact number as required input arguments. Before we move on to an example, it's important that you note the following:

1. In Python 2, the map() function returns a list. In Python 3, however, the function returns a map object which is a generator object. To get the result as a list, the built-in list() function can be called on the map object. i.e. list(map(func, *iterables))
2. The number of arguments to func must be the number of iterables listed.

   Example:

   my_pets = ['alfred', 'tabitha', 'william', 'arla']

   uppered_pets = []

   for pet in my_pets:

      pet_ = pet.upper()

   uppered_pets.append(pet_)

   print(uppered_pets)

   output:

   ['ALFRED', 'TABITHA', 'WILLIAM', 'ARLA']

Filter

While map() passes each element in the iterable through a function and returns the result of all elements having passed through the function, filter(), first of all, requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false. It has the following syntax:

filter(func, iterable)

The following points are to be noted regarding filter():

1. Unlike map(), only one iterable is required.
2. The func argument is required to return a boolean type. If it doesn't, filter simply returns the iterable passed to it. Also, as only one iterable is required, it's implicit that func must only take one argument.
3. filter passes each element in the iterable through func and returns **only** the ones that evaluate to true. I mean, it's right there in the name -- a "filter".

   Example:

```
scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]

def is_A_student(score):

    return score > 75

over_75 = list(filter(is_A_student, scores))

print(over_75)
```

output:

[90, 76, 88, 81]

Reduce:

reduce applies a function **of two arguments** cumulatively to the elements of an iterable, optionally starting with an initial argument. It has the following syntax:

reduce(func, iterable[, initial])

Where func is the function on which each element in the iterable gets cumulatively applied to, and initial is the optional value that gets placed before the elements of the iterable in the calculation, and serves as a default when the iterable is empty. The following should be noted about reduce(): 1. func requires two arguments, the first of which is the first element in iterable (if initial is not supplied) and the second element in iterable. If initial is supplied, then it becomes the first argument to func and the first element in iterable becomes the second element. 2. reduce "reduces" (I know, forgive me) iterable into a single value.

```
from functools import reduce



numbers = [3, 4, 6, 9, 34, 12]
```

output:

68

5)Python File Handling

1)File:

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure file exists otherwise it will get error

2)File operation part-1

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

Demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

o/p:Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

3)File operation part-2

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Ex: Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

f = open("demofile2.txt", "a")

f.write("Now the file has more content!")

f.close()

#open and read the file after the appending:

f = open("demofile2.txt", "r")

print(f.read())

o/p: Hello! Welcome to demofile2.txt

This File is for testing purposes

Good Luck!Now the file has more content!

Delete a File

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

Ex: import os
os.remove("demofile.txt")

4)Handling excel, csv Files

Using xlrd module, one can retrieve information from a spreadsheet. For example, reading, writing or modifying the data can be done in Python. Also, the user might have to go through various sheets and retrieve data based on some criteria or modify some rows and columns and do a lot of work.

xlrd module is used to extract data from a spreadsheet.

Command to install xlrdmodule :
pip install xlrd

The term "csv" stands for "comma separated values" where each row or line contains text based values delimited by commas. In some cases, "semicolon" is also used instead of "comma" to separate values. However, this doesn't make much difference to file format rules and the logic to handle both types of separators remains the same.
CSV file format is most commonly used for maintaining databases and spreadsheets. The first line in a CSV file is most commonly used to define column fields while any other remaining lines are considered rows. This structure allows users to present tabular data using CSV files. CSV files can be edited in any text editor. However, applications like LibreOffice Calc provide advanced editing tools, sort, and filter functions.

6)Python modules and packages:

1) Modules:

The article Module in Python provides you with the information about request, statistics, math and cmath modules. Also, the previous article Modules in Python- Random and System Module provided the information regarding random and system module.

Request Module:

The request module in python allows you to send HTTP requests. Then the HTTP request returns a response object with all the response data (content, encoding, status, etc).

syntax: requests.methodname(parameters)

Method	Description
delete(url, args)	Sends a DELETE request to the specified url
get(url, params, args)	Sends a GET request to the specified url

head(url, args)	Sends a HEAD request to the specified url
patch(url, data, args)	Sends a PATCH request to the specified url
post(url, data, json, args)	Sends a POST request to the specified url
put(url, data, args)	Sends a PUT request to the specified url
request(method, url, args)	Sends a request of the specified method to the specified url

Statistics Module:

Python has a built-in module that you can use to calculate mathematical statistics of numeric data. The statistics module was new in Python 3.4.

Method	Description
statistics.harmonic_mean()	Calculates the harmonic mean (central location) of the given data
statistics.mean()	Calculates the mean (average) of the given data
statistics.median()	Calculates the median (middle value) of the given data
statistics.median_grouped()	Calculates the median of grouped continuous data
statistics.median_high()	Calculates the high median of the given data
statistics.median_low()	Calculates the low median of the given data
statistics.mode()	Calculates the mode (central tendency) of the given numeric or nominal data
statistics.pstdev()	Calculates the standard deviation from an entire population
statistics.stdev()	Calculates the standard deviation from a sample of data
statistics.pvariance()	Calculates the variance of an entire population

statistics.variance()	Calculates the variance from a sample of data

Math Module:

Python has a built-in module that you can use for mathematical tasks. The math module has a set of methods and constants.

Method	Description
math.acos()	Returns the arc cosine of a number
math.acosh()	Returns the inverse hyperbolic cosine of a number
math.asin()	Returns the arc sine of a number
math.asinh()	Returns the inverse hyperbolic sine of a number
math.atan()	Returns the arc tangent of a number in radians
math.atan2()	Returns the arc tangent of y/x in radians
math.atanh()	Returns the inverse hyperbolic tangent of a number
math.ceil()	Rounds a number up to the nearest integer
math.comb()	Returns the number of ways to choose k items from n items without repetition and order
math.copysign()	Returns a float consisting of the value of the first parameter and the sign of the second parameter
math.cos()	Returns the cosine of a number
math.cosh()	Returns the hyperbolic cosine of a number
math.degrees()	Converts an angle from radians to degrees

math.dist()	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point
math.erf()	Returns the error function of a number
math.erfc()	Returns the complementary error function of a number
math.exp()	Returns E raised to the power of x
math.expm1()	Returns $E^x - 1$
math.fabs()	Returns the absolute value of a number
math.factorial()	Returns the factorial of a number
math.floor()	Rounds a number down to the nearest integer
math.fmod()	Returns the remainder of x/y
math.frexp()	Returns the mantissa and the exponent, of a specified number
math.fsum()	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
math.gamma()	Returns the gamma function at x
math.gcd()	Returns the greatest common divisor of two integers
math.hypot()	Returns the Euclidean norm
math.isclose()	Checks whether two values are close to each other, or not
math.isfinite()	Checks whether a number is finite or not
math.isinf()	Checks whether a number is infinite or not
math.isnan()	Checks whether a value is NaN (not a number) or not
math.isqrt()	Rounds a square root number downwards to the nearest integer

math.ldexp()	Returns the inverse of math.frexp() which is x * (2**i) of the given numbers x and i
math.lgamma()	Returns the log gamma value of x
math.log()	Returns the natural logarithm of a number, or the logarithm of number to base
math.log10()	Returns the base-10 logarithm of x
math.log1p()	Returns the natural logarithm of 1+x
math.log2()	Returns the base-2 logarithm of x
math.perm()	Returns the number of ways to choose k items from n items with order and without repetition
math.pow()	Returns the value of x to the power of y
math.prod()	Returns the product of all the elements in an iterable
math.radians()	Converts a degree value into radians
math.remainder()	Returns the closest value that can make numerator completely divisible by the denominator
math.sin()	Returns the sine of a number
math.sinh()	Returns the hyperbolic sine of a number
math.sqrt()	Returns the square root of a number
math.tan()	Returns the tangent of a number
math.tanh()	Returns the hyperbolic tangent of a number
math.trunc()	Returns the truncated integer parts of a number

Math Constants:

Constant	Description
math.e	Returns Euler's number (2.7182…)
math.inf	Returns a floating-point positive infinity
math.nan	Returns a floating-point NaN (Not a Number) value
math.pi	Returns PI (3.1415…)
math.tau	Returns tau (6.2831…)

cMath Module:

Python has a built-in module that you can use for mathematical tasks for complex numbers. The methods in this module accepts int, float, and complex numbers. It even accepts Python objects that has a __complex__() or __float__() method. The methods in this module almost always return a complex number. If the return value can be expressed as a real number, the return value has an imaginary part of 0. The cmath module has a set of methods and constants.

Method	Description
cmath.acos(x)	Returns the arc cosine value of x
cmath.acosh(x)	Returns the hyperbolic arc cosine of x
cmath.asin(x)	Returns the arc sine of x
cmath.asinh(x)	Returns the hyperbolic arc sine of x
cmath.atan(x)	Returns the arc tangent value of x
cmath.atanh(x)	Returns the hyperbolic arctangent value of x
cmath.cos(x)	Returns the cosine of x

cmath.cosh(x)	Returns the hyperbolic cosine of x
cmath.exp(x)	Returns the value of E^x, where E is Euler's number (approximately 2.718281...), and x is the number passed to it
cmath.isclose()	Checks whether two values are close, or not
cmath.isfinite(x)	Checks whether x is a finite number
cmath.isinf(x)	Check whether x is a positive or negative infinty
cmath.isnan(x)	Checks whether x is NaN (not a number)
cmath.log(x[, base])	Returns the logarithm of x to the base
cmath.log10(x)	Returns the base-10 logarithm of x
cmath.phase()	Return the phase of a complex number
cmath.polar()	Convert a complex number to polar coordinates
cmath.rect()	Convert polar coordinates to rectangular form
cmath.sin(x)	Returns the sine of x
cmath.sinh(x)	Returns the hyperbolic sine of x
cmath.sqrt(x)	Returns the square root of x
cmath.tan(x)	Returns the tangent of x
cmath.tanh(x)	Returns the hyperbolic tangent of x

cMath Constant:

Constant	Description

cmath.e	Returns Euler's number (2.7182...)
cmath.inf	Returns a floating-point positive infinity value
cmath.infj	Returns a complex infinity value
cmath.nan	Returns floating-point NaN (Not a Number) value
cmath.nanj	Returns coplextNaN (Not a Number) value
cmath.pi	Returns PI (3.1415...)
cmath.tau	Returns tau (6.2831...)

2) Importing module:

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the `import` keyword to do this. To import our previously defined module `example`, we type the following in the Python prompt.

```
import example
```

This does not import the names of the functions defined in `example` directly in the current symbol table. It only imports the module name `example` there.
Using the module name we can access the function using the dot `.` operator. For example:

example.add(4,5.5)

9.5

Python has tons of standard modules. You can check out the full list of Python standard modules and their use cases. These files are in the Lib directory inside the location where you installed Python.
Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed below..

Python import statement

We can import a module using the `import` statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math

import math
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

Import with renaming

We can import a module by renaming it as follows:

```
# import module by renaming it

import math as m
print("The value of pi is", m.pi)
```

We have renamed the `math` module as `m`. This can save us typing time in some cases.

Note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, and `m.pi` is the correct implementation.

Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module

from math import pi
print("The value of pi is", pi)
```

Here, we imported only the `pi` attribute from the `math` module.

In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

```
>>>from math import pi, e
>>>pi
3.141592653589793
>>>e
2.718281828459045
```

Import all names

We can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math

from math import *
print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in `sys.path`. The search is in this order.

- The current directory.
- `PYTHONPATH` (an environment variable with a list of directories).
- The installation-dependent default directory.

```
>>> import sys
>>>sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add and modify this list to add our own path.

Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named `my_module`.

```
# This module shows the effect of
#  multiple imports and reload

print("This code got executed")
```

Now we see the effect of multiple imports.

```
>>>importmy_module
This code got executed
>>>importmy_module
>>>importmy_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.

Python provides a more efficient way of doing this. We can use the `reload()` function inside the `imp` module to reload a module. We can do it in the following ways:

```
>>>import imp
>>>importmy_module
This code got executed
>>>importmy_module
>>>imp.reload(my_module)
This code got executed
<module 'my_module'from'.\\my_module.py'>
```

The dir() built-in function

We can use the `dir()` function to find out names that are defined inside a module.

For example, we have defined a function `add()` in the module `example` that we had in the beginning.

We can use `dir` in `example` module in the following way:

```
>>>dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']
```

Here, we can see a sorted list of names (along with `add`). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the `__name__` attribute contains the name of the module.

```
>>>import example
>>>example.__name__
'example'
```

All the names defined in our current namespace can be found out using the `dir()` function without any arguments.

```
>>>a = 1
>>>b = "hello"
>>>import math
>>>dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

3) Packages

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has following line of source code −

```
#!/usr/bin/python

def Pots():

print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above −

- Phone/Isdn.py file having function Isdn()
- Phone/G3.py file having function G3()

Now, create one more file __init__.py in Phone directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
#!/usr/bin/python

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()
```

When the above code is executed, it produces the following result −

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

4) Math, Random and OS module:

OS module:

The OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules. This module provides a portable way of using operating system dependent functionality. The *os* and *os.path* modules include many functions to interact with the file system.

```
# Python program to explain os.getcwd() method

# importingos module
import os

# Get the current working
# directory (CWD)
cwd = os.getcwd()

# Print the current working
# directory (CWD)
print("Current working directory:", cwd)
```

Current working directory: /home/nikhil/Desktop/gfg

Random module:

Python has a built-in module that you can use to make random numbers.

The random module has a set of methods:

Method	Description
seed()	Initialize the random number generator
getstate()	Returns the current internal state of the random number generator
setstate()	Restores the internal state of the random number generator

getrandbits()	Returns a number representing the random bits
randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
choices()	Returns a list with a random selection from the given sequence
shuffle()	Takes a sequence and returns the sequence in a random order
sample()	Returns a given sample of a sequence
random()	Returns a random float number between 0 and 1
uniform()	Returns a random float number between two given parameters
triangular()	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters
betavariate()	Returns a random float number between 0 and 1 based on the Beta distribution (used in statistics)

expovariate()	Returns a random float number based on the Exponential distribution (used in statistics)
gammavariate()	Returns a random float number based on the Gamma distribution (used in statistics)
gauss()	Returns a random float number based on the Gaussian distribution (used in probability theories)
lognormvariate()	Returns a random float number based on a log-normal distribution (used in probability theories)
normalvariate()	Returns a random float number based on the normal distribution (used in probability theories)
vonmisesvariate()	Returns a random float number based on the von Mises distribution (used in directional statistics)
paretovariate()	Returns a random float number based on the Pareto distribution (used in probability theories)
weibullvariate()	Returns a random float number based on the Weibull distribution (used in statistics)

Math module:

Python has a built-in module that you can use for mathematical tasks. The math module has a set of methods and constants.

Method	Description
math.acos()	Returns the arc cosine of a number
math.acosh()	Returns the inverse hyperbolic cosine of a number
math.asin()	Returns the arc sine of a number
math.asinh()	Returns the inverse hyperbolic sine of a number
math.atan()	Returns the arc tangent of a number in radians
math.atan2()	Returns the arc tangent of y/x in radians
math.atanh()	Returns the inverse hyperbolic tangent of a number
math.ceil()	Rounds a number up to the nearest integer
math.comb()	Returns the number of ways to choose k items from n items without repetition and order
math.copysign()	Returns a float consisting of the value of the first parameter and the sign of the second parameter
math.cos()	Returns the cosine of a number
math.cosh()	Returns the hyperbolic cosine of a number
math.degrees()	Converts an angle from radians to degrees
math.dist()	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point
math.erf()	Returns the error function of a number

math.erfc()	Returns the complementary error function of a number
math.exp()	Returns E raised to the power of x
math.expm1()	Returns $E^x - 1$
math.fabs()	Returns the absolute value of a number
math.factorial()	Returns the factorial of a number
math.floor()	Rounds a number down to the nearest integer
math.fmod()	Returns the remainder of x/y
math.frexp()	Returns the mantissa and the exponent, of a specified number
math.fsum()	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
math.gamma()	Returns the gamma function at x
math.gcd()	Returns the greatest common divisor of two integers
math.hypot()	Returns the Euclidean norm
math.isclose()	Checks whether two values are close to each other, or not
math.isfinite()	Checks whether a number is finite or not
math.isinf()	Checks whether a number is infinite or not
math.isnan()	Checks whether a value is NaN (not a number) or not
math.isqrt()	Rounds a square root number downwards to the nearest integer
math.ldexp()	Returns the inverse of math.frexp() which is x * (2**i) of the given numbers x and i
math.lgamma()	Returns the log gamma value of x

math.log()	Returns the natural logarithm of a number, or the logarithm of number to base
math.log10()	Returns the base-10 logarithm of x
math.log1p()	Returns the natural logarithm of 1+x
math.log2()	Returns the base-2 logarithm of x
math.perm()	Returns the number of ways to choose k items from n items with order and without repetition
math.pow()	Returns the value of x to the power of y
math.prod()	Returns the product of all the elements in an iterable
math.radians()	Converts a degree value into radians
math.remainder()	Returns the closest value that can make numerator completely divisible by the denominator
math.sin()	Returns the sine of a number
math.sinh()	Returns the hyperbolic sine of a number
math.sqrt()	Returns the square root of a number
math.tan()	Returns the tangent of a number
math.tanh()	Returns the hyperbolic tangent of a number
math.trunc()	Returns the truncated integer parts of a number

Math Constants:

Constant	Description

math.e	Returns Euler's number (2.7182...)
math.inf	Returns a floating-point positive infinity
math.nan	Returns a floating-point NaN (Not a Number) value
math.pi	Returns PI (3.1415...)
math.tau	Returns tau (6.2831...)

7)Classes in python:

1) Classes

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example:

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

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

output:

<class '__mian__.MyClass'>

2)Objects:

Python is an objectoriented programming language.

Almost everything in Python is an object, with its properties and methods.

Create a class:

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

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

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

output:

5

3) Constructor and Deconstructor:

Constructor and Destructor in Python

Constructor is the special function that is automatically executed when an object of a class is created. In Python, there is a special function called “init” which act as a Constructor. It must begin and end with double underscore. This function will act as an ordinary function; but only difference is, it is executed automatically when the object is created. This constructor function can be defined with or without arguments. This method is used to initialize the class variables.

General format of __init__ method (Constructor function)

def __init__(self, [args ……..]):

<statements>

Example : Program to illustrate Constructor

```
class Sample:
def __init__(self, num):
print("Constructor of class Sample...")
self.num=num
print("The value is :", num)
S=Sample(10)
```

The above class “Sample”, has only a constructor with one argument named as num.

When the constructor gets executed, first the print statement, prints the “Constructor of class

Sample….”, then, the passing value to the constructor is assigned to self.num and finally itprints the value passed along with the given string.

The above constructor gets executed automatically, when an object S is created with actual parameter 10. Thus, the Python display the following output.

Constructor of class Sample...

The value is : 10

Class variable defined within constructor keep count of number of objects created with the class.

Example : Program to illustrate class variable to keep count of number of objects created.

```
class Sample:
num=0
def __init__(self, var):
Sample.num+=1
self.var=var
print("The object value is = ", var)
print("The count of object created = ", Sample.num)
S1=Sample(15)
S2=Sample(35)
S3=Sample(45)
```

In the above program, class variable num is shared by all three objects of the class Sample. It is initialized to zero and each time an object is created, the num is incremented by

Since, the variable shared by all objects, change made to num by one object is reflected in other objects as well. Thus the above program produces the output given below.

Output

```
The object value is     =    15
The count of object created   =    1
The object value is     =    35
The count of object created   =    2
The object value is     =    45
The count of object created   =    3
```

Note: class variable is similar to static type in C++

Destructor is also a special method gets executed automatically when an object exit from the scope. It is just opposite to constructor. In Python, __del__( ) method is used as destructor.

Example : Program to illustrate about the __del__( ) method

```
class Sample:
num=0
```

```
def __init__(self, var):

Sample.num+=1

self.var=var

print("The object value is = ", var)

print("The value of class variable is= ", Sample.num)

def __del__(self):

Sample.num-=1

print("Object with value %d is exit from the scope"%self.var)

S1=Sample(15)

S2=Sample(35)

S3=Sample(45)
```

4) Types of methods:

Generally, there are three types of methods in Python:

1. Instance Methods.
2. Class Methods
3. Static Methods

Instance method:

1. Instance Method

This is a very basic and easy method that we use regularly when we create classes in python. If we want to print an instance variable or instance method we must create an object of that required class.

If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.

Note: Instance variables are used with instance methods.

Look at the code below

```
# Instance Method Example in Python

class Student:



    def __init__(self, a, b):
```

```
self.a = a

self.b = b



    def avg(self):

        return (self.a + self.b) / 2



s1 = Student(10, 20)

print( s1.avg() )
```

Copy
Output:

```
15.0
```

2. Class Method

classsmethod() function returns a class method as output for the given function.

Here is the syntax for it:

```
classmethod(function)
```

The classmethod() method takes only a function as an input parameter and converts that into a class method.

There are two ways to create class methods in python:

1. Using classmethod(function)
2. Using @classmethod annotation

A class method can be called either using the class (such as C.f()) or using an instance (such as C().f()). The instance is ignored except for its class. If a class method is called from a derived class, the derived class object is passed as the implied first argument.

As we are working with ClassMethod we use the cls keyword. Class variables are used with class methods.

Look at the code below.

```
# Class Method Implementation in python
```

```
class Student:

    name = 'Student'

    def __init__(self, a, b):

self.a = a

self.b = b



    @classmethod

    def info(cls):

        return cls.name



print(Student.info())
```

Copy

Output:

```
Student
```

3. Static Method

A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.

For example, If you want to print factorial of a number then we don't need to use class variables or instance variables to print the factorial of a number. We just simply pass a number to the static method that we have created and it returns the factorial.

Look at the below code

```
# Static Method Implementation in python

classStudent:

    name ='Student'
```

```
def__init__(self, a, b):

self.a= a

self.b= b



@staticmethod

definfo():

return"This is a student class"



print(Student.info())
```

Copy

Output

```
This a student class
```

8)OOPS in python:

1)Inheritance:

Ever heard of this dialogue from relatives "you look exactly like your father/mother" the reason behind this is called 'inheritance'. From the Programming aspect, It generally means "inheriting or transfer of characteristics from parent to child class without any modification". The new class is called the derived/child class and the one from which it is derived is called a parent/base class.

single Inheritance:

Single level inheritance enables a derived class to inherit characteristics from a single parent class.

Example:

```
1   classemployee1()://This isa parent class
2       def__init__(self, name, age, salary):
3           self.name =name
4           self.age =age
5           self.salary =salary
6
7   classchildemployee(employee1)://This isa child class
8       def__init__(self, name, age, salary,id):
9           self.name =name
10          self.age =age
11          self.salary =salary
12          self.id=id
13  emp1 =employee1('harshit',22,1000)
14
15      print(emp1.age)
```

Output: 22

Explanation:

- Taking the parent class and created a constructor (__init__), class itself is initializing the attributes with parameters('name', 'age' and 'salary').
- Created a child class 'childemployee' which is inheriting the properties from a parent class and finally instantiated objects 'emp1' and 'emp2' against the parameters.

- Finally, I have printed the age of emp1. Well, you can do a hell lot of things like print the whole dictionary or name or salary.

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Example:

```
1   classemployee()://Superclass
2   def__init__(self,name,age,salary):
3   self.name =name
4   self.age =age
5   self.salary =salary
6   classchildemployee1(employee)://First child class
7   def__init__(self,name,age,salary):
8   self.name =name
9   self.age =age
10  self.salary =salary
11
12  classchildemployee2(childemployee1)://Second child class
13  def__init__(self, name, age, salary):
14  self.name =name
15  self.age =age
16  self.salary =salary
17  emp1 =employee('harshit',22,1000)
18  emp2 =childemployee1('arjun',23,2000)
19
20  print(emp1.age)
21  print(emp2.age)
```

Output: 22,23

Explanation:

- It is clearly explained in the code written above, Here I have defined the superclass as employee and child class as childemployee1. Now, childemployee1 acts as a parent for childemployee2.
- I have instantiated two objects 'emp1' and 'emp2' where I am passing the parameters "name", "age", "salary" for emp1 from superclass "employee" and "name", "age, "salary" and "id" from the parent class "childemployee1"

Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Example:

```
1   classemployee():
2   def__init__(self, name, age, salary):     //Hierarchical Inheritance
3   self.name =name
4   self.age =age
5   self.salary =salary
6
7   classchildemployee1(employee):
8   def__init__(self,name,age,salary):
9   self.name =name
10  self.age =age
11  self.salary =salary
12
13  classchildemployee2(employee):
14  def__init__(self, name, age, salary):
15  self.name =name
16  self.age =age
17  self.salary =salary
18  emp1 =employee('harshit',22,1000)
```

```
19                    emp2 =employee('arjun',23,2000)
20
21                    print(emp1.age)
22                    print(emp2.age)
```

Output: 22,23

Explanation:

- In the above example, you can clearly see there are two child class "childemployee1" and "childemployee2". They are inheriting functionalities from a common parent class that is "employee".
- Objects 'emp1' and 'emp2' are instantiated against the parameters 'name', 'age', 'salary'.

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:

```
1     classemployee1()://Parent class
2     def__init__(self, name, age, salary):
3         self.name =name
4         self.age =age
5         self.salary =salary
6
7     classemployee2()://Parent class
8     def__init__(self,name,age,salary,id):
9         self.name =name
10        self.age =age
11        self.salary =salary
12        self.id=id
13
14    classchildemployee(employee1,employee2):
15    def__init__(self, name, age, salary,id):
```

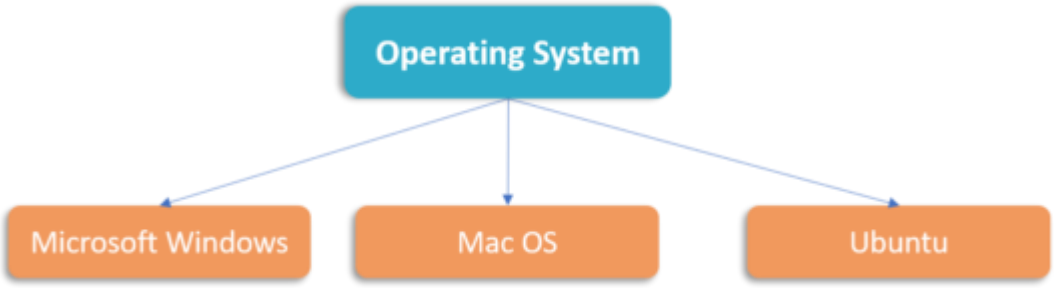
```
16                                  self.name =name
17                                  self.age =age
18                                  self.salary =salary
19                                  self.id=id
20                          emp1 =employee1('harshit',22,1000)
21                          emp2 =employee2('arjun',23,2000,1234)
22
23                                  print(emp1.age)
24                                  print(emp2.id)
```

Output: 22,1234

Explanation: In the above example, I have taken two parent class “employee1” and “employee2”.And a child class “childemployee”, which is inheriting both parent class by instantiating the objects ‘emp1’ and ‘emp2’ against the parameters of parent classes.

2)Polymorphism:

You all must have used GPS for navigating the route, Isn’t it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called ‘polymorphism’. It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, it is a property of an object which allows it to take multiple forms.



Polymorphism is of two types:

- Compile-time Polymorphism
- Run-time Polymorphism

Compile-time Polymorphism:
A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is “method overloading”. Let me show you a quick example of the same.

Example:

```
1   classemployee1():
2       defname(self):
3           print("Harshit is his name")
4       defsalary(self):
5           print("3000 is his salary")
6
7       defage(self):
8           print("22 is his age")
9
10  classemployee2():
11      defname(self):
12          print("Rahul is his name")
13
14      defsalary(self):
15          print("4000 is his salary")
16
17      defage(self):
18          print("23 is his age")
19
20  deffunc(obj)://Method Overloading
21      obj.name()
22      obj.salary()
23      obj.age()
24
25  obj_emp1 =employee1()
```

```
26                                  obj_emp2 =employee2()
27
28                                  func(obj_emp1)
29                                  func(obj_emp2)
```

Output:

Harshit is his name
3000 is his salary
22 is his age
Rahul is his name
4000 is his salary
23 is his age

Explanation:

- In the above Program, I have created two classes 'employee1' and 'employee2' and created functions for both 'name', 'salary' and 'age' and printed the value of the same without taking it from the user.
- Now, welcome to the main part where I have created a function with 'obj' as the parameter and calling all the three functions i.e. 'name', 'age' and 'salary'.
- Later, instantiated objects emp_1 and emp_2 against the two classes and simply called the function. Such type is called method overloading which allows a class to have more than one method under the same name.

Run-time Polymorphism:
A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is "method overriding". Let me show you through an example for a better understanding.

Example:

```
1                                   classemployee():
2                                 def__init__(self,name,age,id,salary):
3                                   self.name =name
4                                    self.age =age
5                                  self.salary =salary
6                                     self.id=id
7                                   defearn(self):
8                                       pass
```

```
 9
10          classchildemployee1(employee):
11
12              defearn(self)://Run-time polymorphism
13                  print("no money")
14
15          classchildemployee2(employee):
16
17              defearn(self):
18                  print("has money")
19
20      c =childemployee1
21      c.earn(employee)
22      d =childemployee2
        d.earn(employee)
```

Output: no money, has money

Explanation: In the above example, I have created two classes 'childemployee1' and 'childemployee2' which are derived from the same base class 'employee'.Here's the catch one did not receive money whereas the other one gets. Now the real question is how did this happen? Well, here if you look closely I created an empty function and used Pass (a statement which is used when you do not want to execute any command or code). Now, Under the two derived classes, I used the same empty function and made use of the print statement as 'no money' and 'has money'.Lastly, created two objects and called the function.

3) Magic methods:

Magic methods are a collection of pre-defined functional method from the python library functions that cannot be declared or called directly. Instead, these functions can be called or invoked by executing some other related code snippet methods. This type of methods are simple to use and implement, as it does not require specific or any kind of extra manual effort from the programmer. Hence it is named as the 'Magic Method'.

What is Python Magic Method?

- Python is an interpreted, object-oriented programming that gives you the ability to write procedural code and/or object-oriented as we know that creating Objects simplifies complicated data structures handling. In addition to that, magic methods eases the ability to create object-oriented programming.
- Before Diving into a magic method, let's understand why they are created in the first place, they are created?
- Below is one example of class one using a magic method and the other is without the magic method. In the first one __init__ magic method is used, which can be used to initialize more than one instance variable in one go. A class Sports is defined as taking two instance variables into account that is name and sport.
- Both instance variables can be defined in one go using the __inti__ magic method. In case 2, the same thing is repeated, but this time we are using a set method to initialize the instance variable. Here for 2 variables, we have to call this method twice.

Here we can see the magic of the magic method; in one go, we can define more than one variable instances.

Code:

```
class Sports():

def __init__(self,name,sport):

self.name = name

self.sport= sport
```

```
def get_name(self):

return self.name

def get_sport(self):

return self.sport

first = Sports('john','Game of Thrones')

print(first.get_name())

print(first.get_sport())
```

Output:

4) Operator overloading:

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

```
# Python program to show use of

# + operator for different purposes.
```

```
print(1 + 2)


# concatenate two strings

print("Geeks"+"For")


# Product two numbers

print(3 * 4)


# Repeat the String

print("Geeks"*4)
```

Output:

```
3
GeeksFor
12
GeeksGeeksGeeksGeeks
```

5) Data hiding(abstraction):

Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don’t know the procedure of how the pin is generated or how the verification is done. This is called ‘abstraction’ from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.

An abstract class cannot be instantiated which simply means you cannot create objects for this type of class. It can only be used for inheriting the functionalities.

Example:

```
1   fromabc importABC,abstractmethod
2   classemployee(ABC):
3   defemp_id(self,id,name,age,salary):    //Abstraction
4   pass
5
6   classchildemployee1(employee):
7   defemp_id(self,id):
8   print("emp_id is 12345")
9
10  emp1 =childemployee1()
11  emp1.emp_id(id)
```

Output: emp_id is 12345

Explanation: As you can see in the above example, we have imported an abstract method and the rest of the program has a parent and a derived class. An object is instantiated for the 'childemployee' base class and functionality of abstract is being used.

6)Encapsulation:

In a raw form, encapsulation basically means binding up of data in a single class. Python does not have any private keyword, unlike Java. A class shouldn't be directly accessed but be prefixed in an underscore.

Let me show you an example for a better understanding.

Example:

```
1   classemployee(object):
2       def__init__(self):
3           self.name =1234
4           self._age =1234
5           self.__salary =1234
6
7       object1 =employee()
8       print(object1.name)
9       print(object1._age)
10      print(object1.__salary)
```

Output:

```
1234
Traceback (most recent call last):
1234
File "C:/Users/Harshit_Kant/PycharmProjects/test1/venv/encapsu.py", line 10, in
print(object1.__salary)
AttributeError: 'employee' object has no attribute '__salary'
```

Explanation: You will get this question what is the underscore and error? Well, python class treats the private variables as(__salary) which can not be accessed directly.

So, I have made use of the setter method which provides indirect access to them in my next example.

Example:

```
classemployee():
def__init__(self):
self.__maxearn =1000000
defearn(self):
print("earning is:{}".format(self.__maxearn))

defsetmaxearn(self,earn)://setter method used foraccesing private class
self.__maxearn =earn

emp1 =employee()
emp1.earn()

emp1.__maxearn =10000
emp1.earn()

emp1.setmaxearn(10000)
emp1.earn()
```

Output:

earning is:1000000,earningis:1000000,earning is:10000

Explanation: Making Use of the setter method provides indirect access to the private class method. Here I have defined a class employee and used a (__maxearn) which is the setter method used here to store the maximum earning of the employee, and a setter function setmaxearn() which is taking price as the parameter.

This is a clear example of encapsulation where we are restricting the access to private class method and then use the setter method to grant access.

INTRODUCTION TO SQL

1. SQL stands for Structured Query Language, which is a standardised language for interacting with RDBMS (Relational Database Management System). Some of the popular relational database example are: MySQL, Oracle, mariaDB, postgreSQL etc.

2. SQL is used to perform C.R.U.D (Create, Retrieve, Update & Delete) operations on relational databases.

3. SQL can also perform administrative tasks on database such as database security, backup, user management etc.

4. We can create databases and tables inside database using SQL.

CONNECTIONS AND QUERIES:-

A Query is a set of instruction given to the database management system, which tells RDBMS what information you would like to get from the database.

Create the Connection

1. Start Enterprise Developer as an administrator. If you need instructions, see *To start Enterprise Developer as an administrator*.
2. From Eclipse, click **Run > Tools > HCO for SQL Server**.
3. From the HCO for SQL Server interface, click **Manage Connections**.
4. Select the **User** Connection Type.
5. From the **Connection List** tab, click **New SQL Server Connection**. This takes you to the **SQL Server Connection** tab.
6. In the **Data Source Name** field, type **HCODemo**.
7. Click **Integrated Windows Authentication** to select it. This means that SQL Server uses your Windows logon credentials to access the database.
8. In the **SQL Server Instance** field, type the name of your SQL Server instance. If you have a default instance, this could be **.** or **(local)**. If you have a named instance, use the combined server name and instance name. For example: **mysqlserver\myinstance**.
9. Click **Confirm Server** to confirm that HCOSS can locate your specified SQL Server Instance.
10. Check **Use a database other than the default database:**.

11. From the **Database** drop-down menu, select the **HCO_Test** destination database that you created in SQL Server Management Studio.

12. Click **Test**. If the test fails, review your field entries, make corrections accordingly, and try again. When you have a successful connection, click **OK**.

13. Click **Save** to save the connection. This takes you back to the **Connection List** tab where you should now see your SQL Server connection listed.

PLSQL/MYSQL

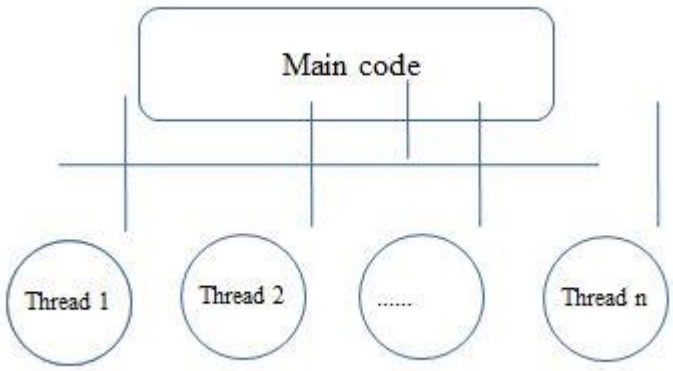
PL/SQL is a block structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks. Pl/SQL stands for "Procedural Language extension of SQL" that is used in Oracle. PL/SQL is integrated with Oracle database (since version 7). The functionalities of PL/SQL usually extended after each release of Oracle database. Although PL/SQL is closely integrated with SQL language, yet it adds some programming constraints that are not available in SQL.

MySQL is an open-source relational database management system used throughout the biggest companies in modern tech. Since its creation, MySQL has established itself as the industry standard for relational database creation and manipulation.To understand the MySQL database, we must break down the underlying branches of tools it belongs: database management systems and SQL Tables.

Multithreading

Multithreading in Python

- It simply means that the execution of two or more threads at the same time.
- The program can be split into multiple parts and these parts execute simultaneously, which in turn increases your program speed, performance and proper use of memory space.



- Each of these threads is responsible for performing different tasks at the same time.

- Multithreading is useful for speeding up program execution and performance improvement, but it is not applied everywhere.
- It can be used only when there is no dependency between threads.
- Every program or process has one thread that will be running always, called the main thread.
- As the diagram shows child threads are created by the main thread.

Regular expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

Create a repository

1. In the upper-right corner of any page, use the drop-down menu, and selectNew **repository**.
2. Type a short, memorable name for your **repository**. ...
3. Optionally, add a description of your **repository**. ...
4. Choose a **repository** visibility. ...
5. Select Initialize this **repository** with a README.
6. Click **Create repository**.

Files State

1. git create branch: create a new branch with git checkout

The fastest way to create a new branch is to actually do it from the git terminal. This way you don't have to use GitHub UI, for example, if you use GitHub for version control. This command actually exists in git, only in a different name – `$ *git checkout*`.
How to create a branch with git checkout
One-line command: `$ *git checkout -b <branch-name> master*`

```
-> commit-test git:(master) $ git checkout -b feature-branch master
-> commit-test git:(feature-branch) $
```

Git tip: just like with commit messages, having a naming convention for git branches is a good best practice to adopt.

2. git force pull: overwrite local with git pull

You find out you've made changes that seemingly conflict with the upstream changes. At this point, you decide to overwrite your changes instead of keeping them, so you do a `$ *git pull*` and you get this error message:

```
-> commit-test git:(master) $ git pull
Updating db40e41..2958dc6
error: Your local changes to the following files would be overwritten by merge:
        README.md
hint:  Please, commit your changes before merging.
fatal: Exiting because of unfinished merge.
```

The problem is that you don't want to commit your changes, you want to overwrite them!

How to overwrite local changes with git pull

1. Stash local changes: `$ *git stash*`
2. Pull changes from remote: `$ *git pull*`

```
-> commit-test git:(master) $ git stash
Updating db40e41..2958dc6
Saved working directory and index state WIP on main: d8fde76 fix(API): remove 'test' end-point
-> commit-test git:(master) $ git pull
Auto-merging README.md
Merge made by the 'recurive' strategy.
README.md      | 1 +
ENDPOINT.js    | 3 ++-
2 files changes, 3 insertions(+), 1 deletions(-)
```

Git tip: if you want to retrieve your changes just do: `$ *git stash apply*`

3. git remove untracked files: delete untracked files from working tree

When having unnecessary files and dirs in your own local copy of a repository, and you want to delete those files, in opposed to just ignore them (with .gitignore), you can use git clean to remove all files which are not tracked by git.

How to remove untracked files and dirs

1. Start with a dry-run to see what will be deleted: `$ *git clean -n -d*`
2. After you are sure, run the git clean command with **-f** flag: `*$ git clean -f -d*`

```
-> commit-test git:(master) $ git clean -n -d
Would remove dontTrackDir/untracked_file1.py
Would remove untracked_file2.py
-> commit-test git:(master) $ git clean -f -d
Removing dontTrackDir/untracked_file1.py
Removing untracked_file2.py
```

Git tip: instead of untracking files, a good practice is to prevent those files from being tracked in the first place by using .gitignore file.

4. git unstage: unstage file(s) from index

When you're adding files ($ git add) to the working tree, you are adding them to the staging area, meaning you are staging them. If you want Git to stop tracking specific files on the working tree, you need to remove them from your stage files (.git/index).

How to unstage file(s) from index

- Keep the file but remove it from the index: `$ *git rm --cached <file-name>*`

```
-> commit-test git:(master) $ git rm –cached unstageMe.js
rm unstageMe.js
```

- To leave the entire working tree untouched, unstage all files (clear your index): `$ *git reset*`

```
-> commit-test git:(master) $ git reset
rm unstageMe.js
```

Git tip: you can also untrack files which already added to git repository based on .gitignore.

5. git undo merge: abort (cancel) a merge after it happened

Sometimes you get in a situation (we've all been there) where you merged branches and realize you need to undo the merge because you don't want to release the code you just merged.

How to abort (cancel) a merge and maintain all committed history

1. Checkout to the master branch: `$ *git checkout master*`
2. Run git log and get the id of the merge commit: `$ *git log --oneline*`
3. Revert merge by commit id: `$ *git revert -m 1 <merge-commit-id>*`
4. Commit the revert and push changes to the remote repo. You can start putting on your poker face and pretend "nothing's happened".

```
-> commit-test git:(master) $ log –oneline
812d761 Merge pull request #524 from datreeio/DAT-1332-resolve-installation-id
b06dee0 feat: added installation event support
8471b2b fix: get organization details from repository object
-> commit-test git:(master) $ revert -m 1 812d761
Revert "Merge pull request #524 from datreeio/DAT-1332-resolve-installation-id"
[master 75b85db] Revert "Merge pull request #524 from datreeio/DAT-1332-resolve-
installation-id"
1 file changed, 1 deletion(-)
-> commit-test git:(master) $ git commit -m "revert merge #524"
-> commit-test git:(master) $ git push
```

Git tip: instead of reverting merge, working with pull requests and setting up or improving your code review process can lower the possibility of a faulty merge.

6. git remove file: remove file(s) from a commit on remote

You wish to delete a file (or files) on remote, maybe because it is deprecated or because this file not supposed to be there in the first place. So, you wonder, what is the protocol to delete files from a remote git repository?

How to remove file(s) from commit

1. Remove your file(s): `$ *git rm <file-A> <file-B> <file-C>*`
2. Commit your changes: `$ *git commit -m "removing files"*`
3. Push your changes to git: `$ *git push*`

```
-> commit-test git:(delete-files) $ git rm deleteMe.js
rm 'deleteMe.js'
-> commit-test git:(delete-files) $ git commit -m "removing files"
[delete-files 75e998e] removing files
1 file changed, 2 deletions(-)
delete mode 100644 deleteMe.js
-> commit-test git:(delete-files) $ git push
```

Git tip: when a file is removed from Git, it doesn't mean it is removed from history. The file will keep "living" in the repository history until the file will be completely deleted.

7. git uncommit: undo the last commit

You made a commit but now you regret it. Maybe you committed secrets by accident – not a good idea – or maybe you want to add more tests to your code changes. These are all legit reasons to undo your last commit.

How to uncommit (undo) the last commit

- To keep the changes from the commit you want to undo: `$ *git reset --soft HEAD^*`
- To destroy the changes from the commit you want to undo: `$ *git reset --hard HEAD^*`

```
-> commit-test git:(undo-commit) $ git commit -m "I will regret this commit"
[undo-commit a7d8ed4] I will regret this commit
1 file changed, 1 insertion(+)
-> commit-test git:(undo-commit) $ git reset –soft HEAD^
-> commit-test git:(undo-commit) $ git status
On branch undo-commit
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

    modified: README.md
```

Git tip: git pre-commit hook is a built-in feature that lets you define scripts that will run automatically before each commit. Use it to reduce the need to cancel commits.

8. git diff between branches

When you are working with multiple git branches, it's important to be able to compare and contrast the differences between two different branches on the same repository. You can do this using the $ git diff command.

How to get the diff between two branches

- Find the diff between the tips of the two branches: `$ *git diff branch_1..branch_2*`
- Produce the diff between two branches from common ancestor commit: `$ *git diff branch_1...branch_2*`
- Comparing files between branches: `$ *git diff branch1:file branch2:file*`

```
-> commit-test git:(diff-me) $ git diff master..diff-me
diff –git a/README.md b/README.md
index b74512d..da1e423 100644
— a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
# commit-test
-Text on “master” branch
+Text on “diff-me” branch
```

Git tip: diff-so-fancy is a great open source solution to make your diffs human readable.

9. git delete tag: remove a tag from branch

In the case of a “buggy” release, you probably don't want someone to accidentally use the release linked to this tag. The best solution is to delete the tag and remove the connection between a release and its co-related tag.

How to delete tag by removing it from branch

1. If you have a remote tag <tag-name> to delete, and your remote is origin, then simply: `$ *git push origin :refs/tags/<tag-name>*`
2. If you also need to delete the tag locally: `$ *git tag -d <tag-name>*`

```
-> commit-test git:(delete-tag) $ git push origin :refs/tags/v1.0.0
To github.com:datreeio/commit-test.git
– [deleted]         v1.0.0
-> commit-test git:(delete-tag) $ git tag -d v1.0.0
Deleted tag ‘v1.0.0’ (was af4d0ea)
```

Git tip: not sure when or why to use tags? Read here to learn more (TL;DR: automatic releasing)

10. git rename branch: change branch name

As I mentioned, having a branch naming convention a good practice and should be adopted as part of your coding standards, and it is especially useful in supporting automation of git workflows. But what to do when you find out your branch name is not aligned with the convention, after already pushing code to the branch? Don't worry, you can still rename your branch.

How to rename branch name after it was created

1. Checkout to the branch you need to rename: `$ *git checkout <old-name>*`
2. Rename branch name locally: `$ *git branch -m <new-name>*`
3. Delete old branch from remote: `$ *git push origin :<old-name> <new-name>*`
4. Reset the upstream branch for the new branch name: `$ *git push origin -u <new-name>*`

```
-> commit-test git:(old-name) $ git branch -m new-name
-> commit-test git:(new-name) $ git push origin :old-name new-name
Total 0 (delta 0), reused 0 (delta 0)
To github.com:datreeio/commit-test.git
- [deleted]             old-name
* [new branch]      new-name -> new-name
-> commit-test git:(new-name) $ git push origin -u new-name
Branch new-name set up to track remote branch new-name from origin.
Everything up-to-date
```

File handling

Exceptions in Python

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.
The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.
We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

```
# import module sys to get the type of exception
```

```
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.
If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).
Here, we print the name of the exception using the exc_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.
Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.")
```

```
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

This program has the same output as the above program.

Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the except clause.
This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause should catch.
A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
try:
   # do something
   pass

except ValueError:
   # handle ValueError exception
   pass

except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass

except:
   # handle all other exceptions
   pass
```

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.
We can optionally pass values to the exception to clarify why that exception was raised.

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt
```

```
>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
...
Enter a positive integer: -2
That is not a positive number!
```

Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside try ran without any errors. For these cases, you can use the optional else keyword with the try statement.

Note: Exceptions in the else clause are not handled by the preceding except clauses.

Example:

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Output

If we pass an odd number:

```
Enter a number: 1
Not an even number!
```

If we pass an even number, the reciprocal is computed and displayed.

```
Enter a number: 4
0.25
```

However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.

```
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

Python try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.
For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.
Here is an example of file operations to illustrate this.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Tkinter provides a powerful object-oriented interface to the Tk **GUI** toolkit.
...
Tkinter Programming

1. Import the **Tkinter** module.
2. Create the **GUI** application main window.
3. Add one or more of the above-mentioned widgets to the **GUI** application.
4. Enter the main event loop to take action against each event triggered by the user.

Beautifulsoup

Beautiful Soup is a Python library for getting data out of HTML, XML, and other markup languages. Say you've found some webpages that display data relevant to your research, such as date or address information, but that do not provide any way of downloading the data directly. Beautiful Soup helps you pull particular content from a webpage, remove the HTML markup, and save the information. It is a tool for web scraping that helps you clean up and parse the documents you have pulled down from the web.

PYTHON FOR DATASCIENCE

Numpy:

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

PANDAS:

Pandas is an open source Python package that is most widely used for data science/data analysis and machine learning tasks. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays. As one of the most popular data wrangling packages, Pandas works well with many other data science modules inside the Python ecosystem, and is typically included in every Python distribution, from those that come with your operating system to commercial vendor distributions like ActiveState's ActivePython.

Matplotlib:

Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy. As such, it offers a viable open source alternative to MATLAB. Developers can also use matplotlib's APIs (Application Programming Interfaces) to embed plots in GUI applications.

Seaborn:

Seaborn is a data visualization library built on top of matplotlib and closely integrated with pandas data structures in Python. Visualization is the central part of **Seaborn** which helps in exploration and understanding of data. ... Visualizing univariate and bivariate distribution.

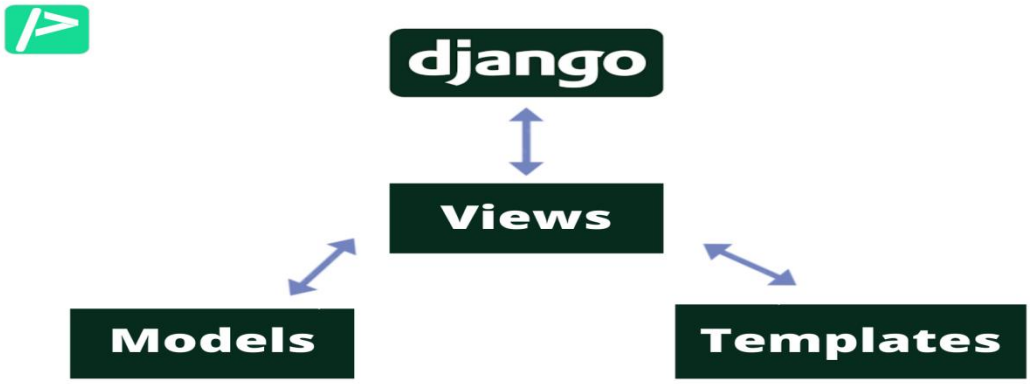
Jupiter Notebook:

Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

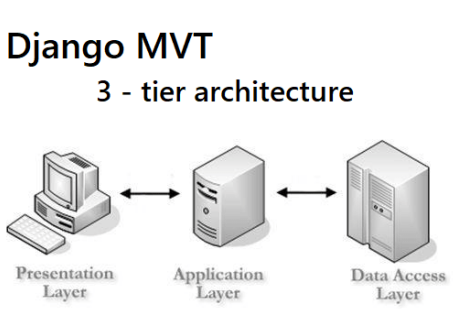
One of *Project Jupyter's* projects is jupyter notebook. **jupyter notebook** is an open-source web app that equips users with tools to write code to achieve various outcomes like data cleaning, data visualization, charting, regression analysis, statistical modeling, machine learning, etc. Jupyter notebook initially only had three core languages, Julia, Python, and R that, when combined, gave birth to the name **Jupyter**. Now, *jupyter notebook* supports over 40 programming languages. These notebooks can easily be shared over github or sent to a colleague via dropbox.

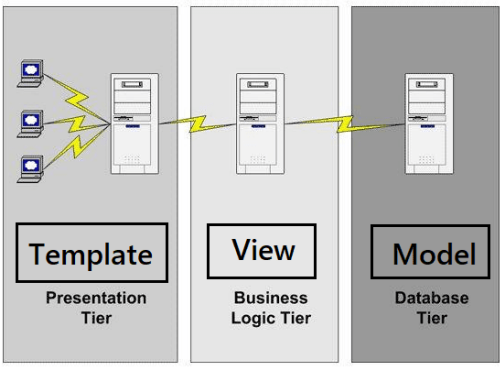
Django urls and views

A request in django first comes to *urls.py* and then goes to the matching function in *views.py*. Python functions in **views.py** takes the web request from **urls.py** and gives the web response to **templates**. It may go to the data access layer in **models.py** as per the *queryset*.



If we look at the 3-tier architecture of an app. Views are like the business logic layer. It is the controller in a typical **MVC** (Model View Controller) design but django has a slighly differnt naming convention called **MVT** (Model View Template) where:
Model is the data access layer,
View is the business logic layer and
Template is the presentation layer.





Django Urls Path

Django has a **urls.py** file under the project by default. It also has a prep-defined path for *admin* app. However, django recommends mapping all resources via another *urls.py* newly created under the app. The below explains it:

mysite -- urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('myapp/', include('myapp.urls')),
]
```

myapp -- urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index),  # app homepage
]
```

Django View Function

The url mapping will redirect request from project urls to app urls and then to the respective view function. A sample view function code may look like this:

```
def index(request):
    return render(request, 'index.html', {})

or,

from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello World")
```

Here, **request** is the url request mapping and calling the view function. **render** combines a given template with a given context dictionary. **{}** denotes the dictionary of values that can be added to the template context.

PYTHON FOR FLASK:

Flask is considered more Pythonic than the Django web framework because in common situations the equivalent Flask web application is more explicit. Flask is also easy to get started with as a beginner because there is little boilerplate code for getting a simple app up and running.

For example, here is a valid "Hello, world!" web application with Flask:

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route('/')
def hello_world():
    return 'Hello, World!'


if __name__ == '__main__':
    app.run()
```

The above code shows "Hello, World!" on localhost port 5000 in a web browser when run with the python app.py command and the Flask library installed.

The equivalent "Hello, World!" web application using the Django web framework would involve significantly more boilerplate code.

Flask was also written several years after Django and therefore learned from the Python community's reactions as the framework evolved. Jökull Sólberg wrote a great piece articulating to this effect in his experience switching between Flask and Django.