SRI INDU COLLEGE OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
HANDS ON TRAINING COURSE
ON
BASICS OF PYTHON PROGRAMMING

Date: From 05.01.2022 To 18-02-2022 (6 Week Course, Only on Saturdays)

COURSE CONTENTS

MODULE -1		
Durations	Topics	Resource Person
Week 1	Basics of python programming What is python programming? Installation and Execution Assignment -1	Mr.K.Raju
Week 2	Applications of python programming An Introduction to Python Installing Python Python – Understanding Data with Visualization Peppering Data	Mr.K.Raju
Week 3	Python 2.x or Python 3.x ? Python interactive: using Python as a calculator Using variables	Mr.K.Raju
MODULE -2		
Durations	Topics	Resource Person
Week 4	Mathematical functions Python scripts (programs) Variables and objects User input, Iterating with indexing Assignment 3	Mr.K.Raju
MODULE -3		
Durations	Topics	Resource Person
Week 5	Understanding Data with Statistics Methods for python programming when to use closures?	Mr.K.Raju

Basics of python programming

1. Python installation

On Linux systems, Python 2.x is already installed. To download Python for Windows and OSx, and for documentation see http://python.org/ It might be a good idea to install the Bethought distribution Canopy that contains already the very useful modulesNumpy, Scipy and Matplotlib: https://www.enthought.com/downloads/

2. Python 2.x or Python 3.x ?

The current version is 3.x Some libraries may not yet be available for version 3, and Linux Ubuntu comes with 2.x as astandard. Many improvements from 3 have been back ported to 2.7. The main differences for basic programming are in the print and input function We will use Python 2.x in this tutorial.

3. Python interactive: using Python as a calculator

Start Python (or IDLE, the Python IDE).A
prompt is showing up:
>>>Display version:>>>help()
Welcome to Python 2.7! This is the online help utility.help>

Help commands:

modules:	available modules
keywords:	list of reserved Python keywords
quit:	leave help

To get help on a keyword, just enter it's name in help.

Simple calculations in Python

```
>>> 3.14*5
15.700000000000001
```

Supported operators:

Operator		Example	Explication
+, - *, /	add, substract, multiply, divide		
%	modulo	25 % 5 = 0 84 % 5 = 4	25/5 = 5, remainder = 0 84/5 = 16, remainder = 4
**	exponent	2**10 = 1024	
//	floor division	84//5 = 16	84/5 = 16, remainder = 4

Take care in Python 2.x if you divide two numbers:
Isn't this strange:

```
>>> 35/65
```

Obviously the result is wrong!
But:

```
>>> 35.0/6
5.833333333333333
>>> 35/6.0
5.833333333333333
```

In the first example, 35 and 6 are interpreted as integer numbers, so integer division is used and the result is an integer.
This uncanny behavior has been abolished in Python 3, where 35/6 gives 5.833333333333333.

In Python 2.x, use floating point numbers (like 3.14, 3.0 etc...) to force floating point division!

Another workaround would be to import the Python 3 like division at the beginning:

```
>>> from __future__ import division
>>> 3/4
0.75
```

Builtin functions:

```
>>> hex(1024)
'0x400'

>>> bin(1024)
'0b10000000000'
```

Expressions:

```
>>> (20.0+4)/64
>>> (2+3)*525
```

4. Using variables

To simplify calculations, values can be stored in variables, and and these can be used as in normal mathematics.

```
>>> a=2.0
>>> b = 3.36
>>> a+b
5.359999999999999
>>> a-b
-1.3599999999999999
>>> a**2 + b**2 15.289599999999998
>>> a>b
False
```

The name of a variable must not be a Python keyword!

Keywords are:

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

5. Mathematical functions

Mathematical functions like square root, sine, cosine and constants like pi etc. are available in Python. To use them it is necessary to import them from the math module:

```
>>> from math import *
>>> sqrt(2)
1.4142135623730951
```

Note:
There is more than one way to import functions from modules. Our simple method imports all functions available in the math module. For more details see appendix.

Other examples using math:

Calculate the perimeter of a circle

```
>>> from math import *
>>> diameter = 5
>>> perimeter = 2 * pi * diameter
>>> perimeter
31.41592653589793
```

Calculate the amplitude of a sine wave:

```
>>> from math import *
>>> Ueff = 230
>>> amplitude = Ueff * sqrt(2)
>>> amplitude
325.2691193458119
```

6. Python scripts (programs)

If you have to do more than a small calculation, it is better to write a script (a program in Python).

This can be done in IDLE, the Python editor.
A good choice is also Geany, a small freeware editor with syntax colouring, from which you can directly start your script.

To write and run a program in IDLE:

- Menu File – New Window
- Write script
- File – Save (name with extension .py, for example myprogram.py)
- Run program: <F5> or Menu Run – Run Module

Take care:

- In Python white spaces are important!
  The indentation of a source code is important!
  A program that is not correctly indented shows either errors or does not what you want!

- Python is case sensitive!
  For example x and X are two different variables.

7. A simple program

This small program calculates the area of a circle:

```
from math import *
d = 10.0                    # diameter
A = pi * d**2 / 4
print "diameter =", d
print "area = ", A
```

Note: everything behind a "#" is a comment.
Comments are important for others to understand what the program does (and for yourself if you look at your program a long time after you wrote it).

8. User input

In the above program the diameter is hard coded in the program.
If the program is started from IDLE or an editor like Geany, this is not really a problem, as it is easy to edit the value if necessary.

In a bigger program this method is not very practical.

This little program in Python 2.7 asks the user for his name and greets him:

```
s = raw_input("What is your name?")
print "HELLO ", s
```

What is your name?Tom
HELLO Tom

Take care:
The raw_input function gives back a string, that means a list of characters. If the input will be used as a number, it must be converted.

9. Variables and objects

In Python, values are stored in objects.
If we do

```
d = 10.0
```

a new object d is created. As we have given it a floating point value (10.0) the object is of type floating point. If we had defined d = 10, d would have been an integer object.

In other programming languages, values are stored in variables. This is not exactly the same as an object, as an object has "methods", that means functions that belong to the object.
For our beginning examples the difference is not important.

There are many object types in Python.

The most important to begin with are:

Object type	Type class name	Description	Example
Integer	int	Signed integer, 32 bit	a = 5
Float	float	Double precision floating point number, 64 bit	b = 3.14
Complex	complex	Complex number	c = 3 + 5j c= complex(3,5)
Character	chr	Single byte character	d = chr(65) d = 'A' d = "A"
String	str	List of characters, text string	e = 'LTAM' e = "LTAM"

10. Input with data conversion

If we use the raw_input function in Python 2.x or the input function in Python 3, the result is always a string. So if we want to input a number, we have to convert from string to number.

```
x = int(raw_input("Input an integer: "))
y = float(raw_input("Input a float: "))
print x, y
```

Now we can modify our program to calculate the area of a circle, so we can input the diameter:

```
""" Calculate area of a circle"""
from math import *
d = float(raw_input("Diameter: "))
A = pi * d** 2 / 4
print "Area = ", A
```

Diameter: 25
Area = 490.873852123

Note:
The text at the beginning of the program is a description of what it does. It is a special comment enclosed in triple quote marks that can spread over several lines.
Every program should have a short description of what it does.

11. While loops

We can use the computer to do tedious tasks, like calculating the square roots of all integers between 0 and 100. In this case we use a while loop:aa

```
0               0.0
1               1.0
2               1.41421356237
3               1.73205080757
.....
98              9.89949493661
99              9.94987437107
100             10.0
READY!
```

The syntax is :

```
while <condition> :
      <....
      block of statements
      ...>
```

The block of statements is executed as long as <condition> is True, in our example as long as i <= 100.

Take care:

- Don't forget the ":" at the end of the while statement
- Don't forget to indent the block that should be executed inside the while loop!

The indentation can be any number of spaces (4 are standard), but it must be consistent for the whole block.

Avoid endless loops!

In the following example the loop runs infinitely, as the condition is always true:

```
i = 0
while i<= 5 :
    print i
```

The only way to stop it is by pressing <Ctrl>-C.

Examples of conditions:

Example	
x == 3	True if x = 3
x != 5	True if x is not equal to 5
x < 5 x > 5	
x <= 5 x >= 5	

Note:
i = i +1 can be written in a shorter and more "Pythonic" way as i += 1

12. Testing conditions: if, elif, else

Sometimes it is necessary to test a condition and to do different things, depending on the condition.
Examples: avoiding division by zero, branching in a menu structure etc.

The following program greets the user with "Hello Tom", if the name he inputs is Tom:

```
s = raw_input ("Input your name: ")

if s == "Tom":
    print "HELLO ", s
```

Note the indentation and the ":" behind the if statement!

The above program can be extended to do something if the testing condition is not true:

```
s = raw_input ("Input your name: ")
if s == "Tom":
    print "Hello ", s
else:
    print "Hello unknown"
```

It is possible to test more than one condition using the elif statement:

```
s = raw_input ("Input your name: ")
if s == "Tom":
    print "Hello ", s
elif s == "Carmen":
    print "I'm so glad to see you ", s
elif s == "Sonia":
    print "I didn't expect you ",s
else:
    print "Hello unknown"
```

Note the indentation and the ":" behind the if, elif and else statements!

13. Tuples

In Python, variables can be grouped together under one name. There are different ways to do this, and one is to use tuples.

Tuples make sense for small collections of data, e.g. for coordinates:

```
(x,y) = (5, 3)
coordinates = (x,y)
print coordinates

dimensions = (8, 5.0, 3.14)
print dimensions
print dimensions[0]
print dimensions[1]
print dimensions[2]
```

(5, 3)
(8, 5.0, 3.14)
8
5.0
3.14

Note:
The brackets may be omitted, so it doesn't matter if you write x, y or (x, y)

14. Lists (arrays)

Lists are ordered sequences of objects.
It can for example be very practical to put many measured values, or names of an address book, into a list, so they can be accessed by one common name.

```
nameslist = ["Sam", "Lisy", "Pit"]
numberslist = [1, 2, 3.14]
mixedlist = ["ham", 'eggs', 3.14, 5]
```

Note:
Unlike other programming languages Python's arrays may contain different types of objects in one list.

New elements can be appended to a list:

```
a=[0,1,2]
print a

a.append(5)
a.append( "Zapzoo")
print a
```

[0, 1, 2]
[0, 1, 2, 5, 'Zapzoo']

An empty list can be created this way:

```
x=[]
```

Sometimes we need an array that is initialized with zero values.

This is done with:

```
y= [0]*10      # array of integers with 10 zero elements
z = [0.0]*20   # array of floats with 20 zero elements
```

The number of elements can be determined with the len (length) function:

```
a=[0,1,2]
print len(a)
```

3

The elements of a list can be accessed one by one using an index:

```
mylist = ["black", "red", "orange"]
print mylist[0]
print mylist[1]
print mylist[2]
```

black
red
orange

15. Range: producing lists of integer numbers

Often you need a regularly spread list of numbers from a beginning value to an end value.

This is done by the range function:

```
""" range gives a list of int numbers
    note that end value is NOT included! """

r1 = range(11)          # 0...10
print r1                 # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

r2 = range(5,16)        # 5...15
print r2                 # [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

r3 = range(4,21,2)      # 4...20  step 2
print r3                 # [4, 6, 8, 10, 12, 14, 16, 18, 20]

r4 = range(15, 4, -5)   # 15....5 step -5
print r4                 # [15, 10, 5]
```

The general syntax is

```
range (<startvalue>, <endvalue>, <stepsize>)
```

Take care:

- A strange (and somewhat illogical) detail of the range function is that the end value is excluded from the resulting list!
- The range function only works for integers!

16. Producing lists of floating point numbers

If you need floating point numbers, use linspace from the Numpy module, a package that is very useful for technical and scientific applications. This package must be installed first, it is available at http://www.numpy.org/
Don't forget to import the module in your script!

Note:
Here we use a slightly different method of import that avoids confusion between names of variables and numpy funtions. There are 3 ways to import functions from a module, see appendix.

```
""" for floating point numbers use linspace and logspace from numpy!"""
import numpy as np
r5 = np.linspace(0,2,9)
print r5
```

```
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.  ]
```

The syntax for linspace is

```
linspace ( <startvalue>, <stopvalue>, <number of values> )
```

The next example gives 9 logarithmically spaced values between 100 = 10^2 and 1000 = 10^3:

```
r6 = np.logspace(2, 3, 9)
print r6
```

```
[  100.          133.35214322  177.827941    237.13737057  316.22776602
   421.69650343  562.34132519  749.89420933  1000.        ]
```

17. Iterating through a list: the for loop

If we have to do something with all the elements of a list (or another sequence like a tuple etc.) one after the other, we use a for loop.

The example uses the names of a list of names, one after one:

```
mynames = [ "Sam", "Pit", "Misch" ]

for n in mynames:
    print "HELLO ", n
```

```
HELLO  Sam
HELLO  Pit
HELLO  Misch
```

This can also be done with numbers:

```
from math import *
for i in range (0, 5):
    print i, "\t", sqrt(i)
```

```
0       0.0
1       1.0
2       1.41421356237
3       1.73205080757
4       2.0
```

Notes:

- Python's for loop is somewhat different of the for ... next loops of other programming languages. in principle it can iterate through anything that can be cut into slices. So it can be used on lists of numbers, lists of text,mixed lists, strings, tuples etc.

- In Python the for ... next construction is often not to be missed, if we think in a "Pythonic" way. Example: if we need to calculate a lot of values, it is not a good idea to use a for loop, as this is very time consuming. It is better to use the Numpy module that provides array functions that can calculate a lot of values inone bunch (see below).

18. Iterating with indexing

Sometimes you want to iterate through a list and have access to the index (the numbering) of the items of the list.

The following example uses a list of colour codes for electronic parts and prints their index and the colour. As the colours list is well ordered, the index is also the colour value.

```
""" Dispay resistor colour code values"""
colours = [ "black", "brown", "red", "orange", "yellow",
            "green", "blue", "violet", "grey","white" ]

cv = list (enumerate (colours))

for c in cv:
    print c[0], "\t", c[1]
```

The list(enumerate (...)) function gives back a list of tuples cv that contain each an index (the numbering) and the color value as text. If we print this we see

[(0, 'black'), (1, 'brown'), (2, 'red'), (3, 'orange'), (4, 'yellow'), (5,
'green'), (6, 'blue'), (7, 'violet'), (8, 'grey'), (9, 'white')]

Now we iterate on this, so we get the different tuples one after the other.

From these tuples we print c[0], the index and c[1], the colour text, separated by a tab.

So as a result we get:

```
0       black
1       brown
2       red
...
8       grey
9       white
```

19. Functions

It is a good idea to put pieces of code that do a clearly defined task into separate blocks that are called functions. Functions must be defined before they are used.

Once they are defined, they can be used like native Python statements.

A very simple example calculates area and perimeter of a rectangle:

```
# function definitions
def area(b, h):
""" calculate area of a rectangle"""
A = b * h return A

def perimeter(b, h):
""" calulates perimeter of a rectangle"""
P = 2 * (b+ h) return P

# main program using defined functions
width = 5
height = 3
print "Area = ", area(width, height)
print "Perimeter = ", perimeter(width, height)
```

The syntax of a function definition is:

```
def <function_name(<argument1>, <argument2>, ……):
      <statements>
      ....
      return <returnvalue(s)>
```

The arguments are the values passed to the function.

the return value is the value that the function gives back to the calling program statement.

Dont't forget the ":" and the indentation !

A function can return more than one value:

```
# function definition
def area_and_perimeter (b, h):
    A = b * h
    P = 2 * (b+h)
    return A, P

# main program using defined function
ar, per = area_and_perimeter ( 4, 3)
print ar
print per
```

Here the return values are returned as a tuple.

If the function doesn't need to return a value, the return statement can simply be omitted.

Example:

```
# function definition
def greeting():
    print "HELLO"


# main program using defined functions
greeting()
```

One good thing about functions is that they can be easily reused in another program.

Notes:

- Functions that are used often can be placed in a separate file called a module. Once this module is importedthe functions can be used in the program.
- It is possible to pass a variable number of arguments to a function.

For details see here: http://en.wikibooks.org/wiki/Python_Programming/Functions

- It is possible to pass named variables to a function

20. Avoiding for loops: vector functions

For loops tend to get slow if there are many iterations to do.

They are not necessary for calculations on numbers, if the **Numpy** module is used. It can befound here http://www.numpy.org/ and must be installed before using it.

In this example we get 100 values of a sine function in one line of code:

```
import numpy as np

# calculate 100 values for x and y without a for loop
x = np.linspace(0, 2* np.pi, 100)
y = np.sin(x)

print x
print y
```

21. Diagrams

Once you have calculated the many function values, it would be nice to display them in adiagram. This is very simple if you use Matplotlib, the standard Python plotting library.

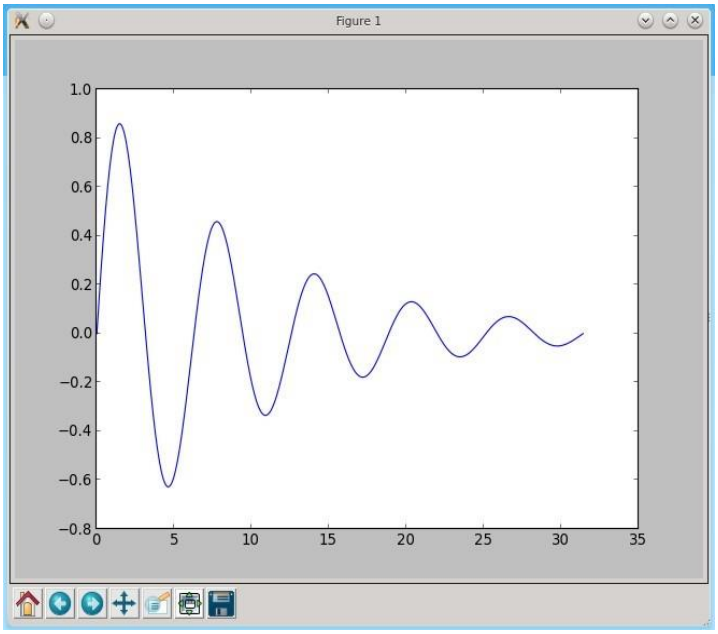
Matplotlib can be found here: http://matplotlib.org/downloads

The following program calculates the values of a function and draws a diagram:

```
from numpy import linspace, sin, exp, pi
import matplotlib.pyplot as mp

# calculate 500 values for x and y without a for loop
x = linspace(0, 10*pi, 500)
y = sin(x) * exp(-x/10)

# make diagram
mp.plot(x,y)
mp.show()
```



Notes:

- Matplotlib offers much more than this, see online documentation.
- There are two ways to use Matplotlib: a simple functional way that we have just used, and a morecomplicated object oriented way, that allows for example to embed a diagram into a GUI.

22. What next?

This tutorial covered just a minimal part of the Python basics. There are many, many interesting possibilities to discover:

Object oriented programming, programs with a graphical user interface (GUI), connecting to hardware, signal processing, image and sound processing etc. etc.

The Python package index is a good place to look for interesting modules:

https://pypi.python.org/pypi

Appendix

Importing functions from a module
Three ways to import functions:

1.the simplest way: import everything from a moduleadvantage: simple usage e.g. of math functions disadvantage: risk of naming conflicts when a variable has the same name as a module function

```
from numpy import *
print sin(pi/4)

# With this import method the following would give an error:
#sin = 5              # naming conflict!
#print sin(pi/4)
```

2.import module under an alias name that is short enough to enhance code clarityadvantage: it is clear to see which function belongs to which module

```
import numpy as np
print np.sin(np.pi/4)
```

3.import only the functions that are needed advantage: simple usage e.g. of math functions naming conflict possible, but less probable than with

1.disadvantage: you must keep track of all the used functions and adapt the import statement if anew function is used

```
from numpy import linspace, sin,
exp, piprint sin(pi/4)
```

Python Directory

If there are a large number of files to handle in our Python program, we can arrange our code within different directories to make things more manageable. A directory or folder is a collection of files and subdirectories. Python has the os module that provides us with many useful methods to work with directories (and files as well).

Get Current Directory

We can get the present working directory using the getcwd() method of the os module.
This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

```
>>> import os

>>> os.getcwd()
'C:\\Program Files\\PyScripter'

>>> os.getcwdb()
b'C:\\Program Files\\PyScripter'
```

The extra backslash implies an escape sequence. The print() function will render this properly.

```
>>> print(os.getcwd())
```

C:\Program Files\PyScripter

Changing Directory

We can change the current working directory by using the chdir() method.
The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash / or the backward-slash \ to separate the path elements.
It is safer to use an escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())
C:\Python33
```

List Directories and Files

All files and sub-directories inside a directory can be retrieved using the listdir() method.
This method takes in a path and returns a list of subdirectories and files in that path. If no path is specified, it returns the list of subdirectories and files from the current working directory.

```
>>> print(os.getcwd())
C:\Python33

>>> os.listdir()
['DLLs',
'Doc',
'include',
'Lib',
'libs',
'LICENSE.txt',
'NEWS.txt',
'python.exe',
'pythonw.exe',
'README.txt',
'Scripts',
'tcl',
'Tools']

>>> os.listdir('G:\\')
['$RECYCLE.BIN',
'Movies',
'Music',
'Photos',
'Series',
'System Volume Information']
```

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`.

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
```

Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use the `class` keyword to define an empty class `Parrot`. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is an object of class `Parrot`.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name `Parrot`. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the `__init__` method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the `Parrot` class. Here, `blu` and `woo` are references (value) to our new objects.

We can access the class attribute using `class__.species`. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using `blu.name` and `blu.age`. However, instance attributes are different for every instance of a class.

To learn more about classes and objects, go to Python Classes and Objects

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance methods because they are called on an instance object i.e `blu`.

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")
```

```
    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In the above program, we created two classes i.e. `Bird` (parent class) and `Penguin` (child class). The child class inherits the functions of parent class. We can see this from the `swim()` method.

Again, the child class modified the behavior of the parent class. We can see this from the `whoisThis()` method.

Furthermore, we extend the functions of the parent class, by creating a new `run()` method.

Additionally, we use the `super()` function inside the `__init__()` method. This allows us to run the `__init__()` method of the parent class inside the child clas

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single `_` or double `__`.

Example 4: Data Encapsulation in Python

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
```

```
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a `Computer` class.

We used `__init__()` method to store the maximum selling price of `Computer`. Here, notice the code `c.__maxprice = 1000`

Here, we have tried to modify the value of `__maxprice` outside of the class. However, since `__maxprice` is a private variable, this modification is not seen on the output.

As shown, to change the value, we have to use a setter function i.e `setMaxPrice()` which takes price as a parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
k

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Output

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes `Parrot` and `Penguin`. Each of them have a common `fly()` method. However, their functions are different.

To use polymorphism, we created a common interface i.e `flying_test()` function that takes any object and calls the object's `fly()` method. Thus, when we passed the `blu` and `peggy` objects in the `flying_test()` function, it ran effectively.

Iterators in Python

Iterators are everywhere in Python. They are elegantly implemented within `for` loops, comprehensions, generators etc. but are hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Iterating Through an Iterator

We use the `next()` function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise the `StopIteration` Exception. Following is an example.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

Output

```
4
7
0
3
Traceback (most recent call last):
  File "<string>", line 24, in <module>
    next(my_iter)
StopIteration
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

Generators in Python

There is a lot of work in building an iterator in Python. We have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, and raise `StopIteration` when there are no values to be returned.
This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Create Generators in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a `yield` statement instead of a `return` statement. If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function. Both `yield` and `return` will return some value from a function. The difference is that while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and Normal function

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

- Generator function contains one or more `yield` statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named `my_gen()` with several `yield` statements.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1 >>> # Once the function yields, the function is paused and the control is transferred to the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>> next(a)
This is printed second
2 >>> next(a)
This is printed at last
3 >>> # Finally, when the function terminates, StopIteration is raised automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that the value of variable `n` is remembered between each call.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like `a = my_gen()`.

One final thing to note is that we can use generators with for loops directly.

This is because a `for` loop takes an iterator and iterates over it using `next()` function. It automatically ends when `StopIteration` is raised. Check here to know how a for loop is actually implemented in Python.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n


# Using for loop
for item in my_gen():
    print(item)
```

Python Closures

Nonlocal variable in a nested function

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

In Python, these non-local variables are read-only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

Output

```
Hello
```

We can see that the nested `printer()` function was able to access the non-local `msg` variable of the enclosing function.

Defining a Closure Function

In the example above, what would happen if the last line of the function `print_msg()` returned the `printer()` function instead of calling it? This means the function was defined as follows:

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    return printer  # returns the nested function


# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

Output

```
Hello
```

That's unusual.

The `print_msg()` function was called with the string `"Hello"` and the returned function was bound to the name `another`. On calling `another()`, the message was still remembered although we had already finished executing the `print_msg()` function.

This technique by which some data (`"Hello` in this case) gets attached to the code is called **closure in Python**.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Try running the following in the Python shell to see the output.

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

Here, the returned function still works even when the original function was deleted.

When do we have closures?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

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

When to use closures?

So what are closures good for?

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solution. But when the number of attributes and methods get larger, it's better to implement a class.

Python RegEx

In this tutorial, you will learn about regular expressions (RegEx), and use Python's re module to work with RegEx (with the help of examples).

A **Reg**ular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,
^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with a and ending with s**.

A pattern defined using RegEx can be used to match against a string.

Expression	String	Matched?
	abs	No match
	alias	Match
^a...s$	abyss	Match
	Alias	No match
	An abacus	No match

Python has a module named `re` to work with RegEx. Here's an example:

```
import re

pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)

if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

Here, we used `re.match()` function to search `pattern` within the `test_string`. The method returns a match object if the search is successful. If not, it returns `None`.

There are other several functions defined in the `re` module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.