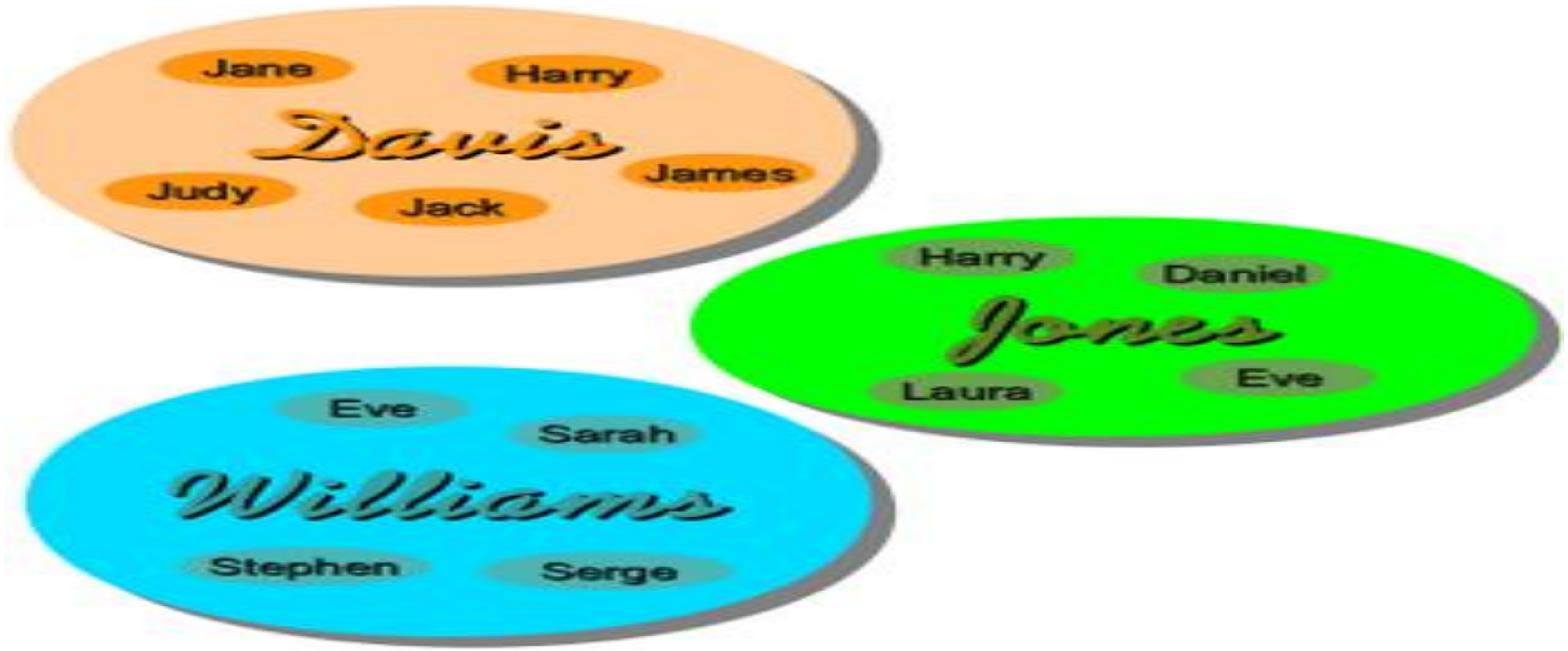
Python

Namespaces

- Taking cue from the gentleman to the left, I would like to say "Python is a PHUNNY Language"
- A variable "**var1**" can be a integer, and then immediately a float, and then even a class or function.
- A function can have a name ***YoYoHoneySingh inside it, while another function can have the same name as well.***
- All these variables are actually objects, the way to reach these objects are through names.
- *Now, where do you keep all these contradictions without causing a conflict.*
- *We store them in NAMSEPACES.*

NAMESPACES

- As we take a peep into namespaces, we need to understand what happens to all those variables(or rather objects) that we ourselves create, or import through modules.
- Its something like this



Namespaces

- A namespace is thus a logical concept, think of it like different tumblers holding different names.
- Modules and namespaces go hand-in-hand.
- A module , whenever imported, gets its own namespace within which it exists.
- Each namespace is completely isolated. Two modules can have the same name within them.
- There can be a module named ***Integer***, and another module named ***Float.***
- *Both can have the same method **Add()** within them.*
- *When we invoke the Add() which resides within Integer, we invoke Integer.Add()*
- *To invoke Add() which resides within Float, we invoke Float.Add()*
- *Whenever you run a simple Python script, the interpreter treats it as module called __main__, which gets its own namespace. The builtin functions that you would use also live in a module called __builtin__ and have their own namespace.*

Modules and Namespaces

A namespace, is obviously enough, a space that holds a bunch of names. The standard Python tutorials say that they are a mapping from names to objects. Think of it as a big list of all the names that you've defined, either explicitly or by importing from modules. It's not something than you have to create, it's created whenever necessary. To understand namespaces, we would also need to understand the concepts of **Modules(Coming down to the details in a while**). However, for the moment, think of it as a file which contains your Python code.The code can be anything :- Functions, classes, plain code body, or even a Module. Each module gets its own namespace.

So you can't have two functions or two classes in the same Module with the same name.

However each namespace is also completely isolated. So two modules can have the same names within them. You can have a module called Integer and a module called FloatingPoint and both could have a function named add(). Once you import the module into your script, you can access the names by prefixing them with the module name: **FloatingPoint.add() and Integer.add()**.

Whenever you run a simple Python script, the interpreter treats it as module called __main__, which gets its own namespace. The builtin functions that you would use also live in a module called __builtin__ and have their own namespace.

SCOPE

- A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
- The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

MODULES

- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example :-

The Python code for a module named **aname** normally resides in a file namedaname.py. Here is an example of a simple module, support.py −

```
def  print_func(par):
   print(“Hello :”, par)
```

Importing Modules

You can use any Python source file as a module by executing an import statement in some other Python source file. The **import** has the following syntax −

MODULES

```
import support as sp
 sp.print_func('Zara')
```

- Remember in this case, we get access to the module **support's** namespace.
- All work will be done inside the modules namespace. Lets   put in another example:-

```
import math
def ceil(a1,a2):
        a = a1-a2
        return a
a = math.ceil(1.5)
b = ceil(1,22)
```

- Can you explain whats the output?

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

MODULES

The from…import Statement:

```
from  support import print_func
print_func('Zara')
```

This imports a name (or a few, separated by commas) from a module's namespace directly into the program's. To use the name you imported, you no longer have to use a prefix, just the name directly. This can be useful if you know for certain you'll only need to use a few names. The downside is that you can't use the name you imported for something else in your own program. For example, you could use add() instead of Integer.add(), but if your program has an add() function, you'll lose access to the Integer's add() function.

Example :-

```
>>> from functions import multiply
>>> multiply(1,2)
4
>>> def multiply(a,b):
        return(a/b)
>>> a = multiply(4,2)
>>> print(a)
2.0
```

MODULES

The from import* function

```
from  modname import*
```

- This imports all the names from inside the **module** directly inside the module's namespace.
- Generally not a good idea as it leads to 'namespace pollution'.

The reload function

- Once a module is imported, the code inside the modules are improrted only once.
- After the first import, the later imports simply do nothing.
- This is by design. Actually import statements are very heavy and expensive(resource extensive)
- To do so , use the reload function.
- Reload, in a sense refreshes the module .
- The format of the reload function is reload(modulename)
- To run the reload function,oops, its now a method, we need to import the **imp** module.

Directory function

- *The **dir()** function is used to find all components/attributes inside the module.*
- The **dir()** built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

```
>>> import math
>>> print(dir(math))
```

- *['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']*
- Here, the special string variable __name__ is the module's name, and __file__ is the filename from which the module was loaded.

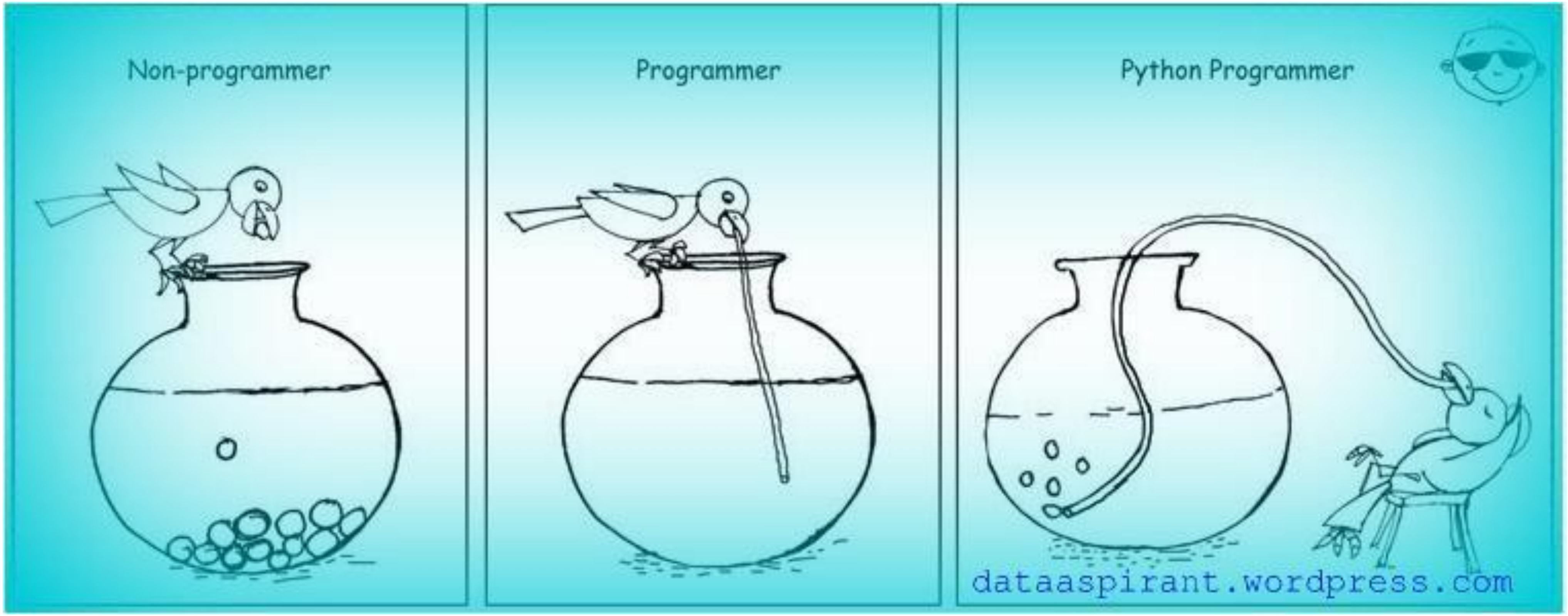
LOCATING MODULES

- When we import a module, the Python interpreter searches for the module in the following sequences −
- The current directory.
- If the module is not found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python3/.
- The module search path is stored in the system module sys as the **sys.path**variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

<u>The **PYTHONPATH** variable:</u>

- The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.
- Here is a typical PYTHONPATH from a Windows system and one from a Unix system −
- <u>Windows : set PYTHONPATH = c:\python34\lib;</u>
- <u>Unix : set PYTHONPATH = /usr/local/lib/python</u>
- Last priority is for searching in the PATH variable

PYTHON PACKAGES



PYTHON PACKAGES

- A Python package is just a collection of Python modules.
- A package is just a way of collecting related modules together within a single-tree like structure.

Creating a Package

- Store all your modules in a folder.
- The folder name is the Package Name.
- Create **__init__** in the same folder with import statements of all the modules.
- Import package in other programs with import package name command.

FILE OPERATIONS

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization. (Gerald Weinberg)

PYTHON FILE MANAGEMENT

- Till now we have seen how to read/write/print from standard input/standard output.
- A quick recap, we read from the keyboard using the function input().
- We print using the print() function.
- We all use print formatting, to print various types of datatypes like int,var etc.
- Now how do we read files and write into it.Here's how we start a new chapter all together.

The open() function :-

- Before reading or writing a file, we have to open it using python's built in function **open()** .
- We open a file by using the syntax ***fp = open(file_name [,access mode] [,buffering])***
- **fp** is a file object, which is returned whenever you open the file to read/write.

PYTHON FILE MANAGEMENT

Till now we have seen how to read/write/print from standard input/standard output.

- A quick recap, we read from the keyboard using the function raw_input().
- We print using the print() function.
- We all use print formatting, to print various types of datatypes like int,var etc.
- Now how do we read files and write into it.Here's how we start a new chapter all together.
- The **open()** function :-
- Before reading or writing a file, we have to open it using python's built in function **open()** .
- We open a file by using the syntax ***fp = open(file_name [,access mode] [,buffering])***
- **fp** is a file object, which is returned whenever you open the file to read/write.
- The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).

PYTHON FILE MANAGEMENT

- Buffering is a concept being parked aside for now.
- It's a optional parameter. Usually useful when we have to read large files, those going upto sizes of GB's.
- Buffering rations the amount of input that is read at one shot, much like a nutrionist would ration food for a person looking to lose weight ☺
- If Buffering is enabled, instead of trying to read an entire file at one shot, the huge file is read in chunks of bytes in memory. Otherwise the RAM would have gone Mad . Think of a 24 GB file being read at one shot.

  RAM Naam Satya Hain!! ☺
- '0' to switch buffering off (only allowed in binary mode) '1' to select line buffering (only usable in text mode) 'integer > 1' to indicate the size of a fixed-size chunk buffer

PYTHON FILE MANAGEMENT

The read() method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

PYTHON FILE MANAGEMENT- Different file modes

S.No.	Mode & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

PYTHON FILE MANAGEMENT- Different file modes

S.No.	Mode & Description
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

PYTHON FILE MANAGEMENT- FILE OBJECT ATTRIBUTES

- Once a file is opened and you have one *file* object, you can get various information related to that file.
- Here is a list of all the attributes related to a file object −

Sl.No	Attribute & Description
1	**file.closed** Returns True, if a file is closed, false otherwise.
2	**file.mode** Returns the access mode with which file was opened
3	**file.name** Returns the name of the file

PYTHON FILE MANAGEMENT- Reading & writing

- The **read()** method reads a string from an open file. It is important to note that Python strings can have binary data, apart from text data.
- Syntax is fo.read([count of bytes])
  - Count refers to the number of bytes that has to be read.
  - Lets refer the file foo.txt, and read 10 bytes out of it.

File Positions :-

- To understand where the file has been read upto, we use the tell() method.
- The tell() method, indicates where the current file pointer rests at.
- The next read or write, will begin at that position.
- There is another method called **seek()** which changes the current file position.
- Its syntax is **fo.seek(offset[,from])**

```
>>> fo = open("foo.txt","r")
>>> fo.read(3)
'Pyt‘
>>> fo.tell()
3
```

FILE OPERATIONS

The writeline() method:

- The method **writelines()** writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no

Syntax :-

fileObject.writelines(sequence)

```
# Assuming a file, which already has the following 5 lines wriiten
# This is 1st line
# This is 2nd line
# This is 3rd line
# This is 4th line
# This is 5th line
We need to add 2 more lines, but at one shot :
        fo = open("foo.txt","a+")
        fo.seek(0,0)
        >>> list= ['# This is 6th line', "\n",'# This is 7th line']
        >>> fo.writelines(list)
        >>> fo.tell()
        170
        >>> fo.close()
```

PYTHON FILE MANAGEMENT- FILE OPERATIONS

Renaming & Deleting files :

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module, you need to import it first and then you can call any related functions.

The rename() Method

- The rename() method takes two arguments, the current filename and the new filename.

Syntax :-

Import os

- os.rename(current_file_name, new_file_name)

>>> os.rename("sample.txt",“sample1.txt")

The remove() Method

Use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax :-

os.remove(file_name)

PYTHON FILE MANAGEMENT-Directory methods

Mkdir() method :-

- You can use the **mkdir()** method of the **os** module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax

- os.mkdir("newdir")

chdir() Method :-

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

- os.chdir("newdir")

Stimulants

PYTHON FILE MANAGEMENT- Reading & writing

1. To open a file c:\scores.txt for reading, we use
   a) infile = open("c:\scores.txt", "r")
   b) infile = open(file = "c:\scores.txt", "r")
   c) infile = open(file = "c:\\scores.txt", "r")
2. To open a file c:\scores.txt for writing, we use
   a) outfile = open("c:\scores.txt", "w")
   b) outfile = open("c:\\scores.txt", "w")
   c) outfile = open(file = "c:\scores.txt", "w")
   d) outfile = open(file = "c:\\scores.txt", "w")
3. To read two characters from a file object infile, we use
   a) infile.read(2)
   b) infile.read()
   c) infile.readline()
   d) infile.readlines()

PYTHON FILE MANAGEMENT- Reading & writing

4. What is the output?

```
f = None
for i in range (5):
    with open("data.txt", "w") as f:
        if i > 2:
            break
print(f.closed)
```

a) True
b) False
c) None
d) Error

Note : opening a file with *with* will ensure that the file is closed as well.

5. What is the use of tell() method in python?

a) tells you the current position within the file
b) tells you the end position within the file
c) tells you the file is opened or not
d) none of the mentioned

PYTHON FILE MANAGEMENT- Reading & writing

6. What is the current syntax of rename() a file?

a) rename(current_file_name, new_file_name)
b) rename(new_file_name, current_file_name,)
c) rename(()(current_file_name, new_file_name))
d) none of the mentioned

7. What is the output of the following lines of code ?

```
fo = open("foo.txt", "rw+")
print ("Name of the file: ", fo.name)
```

8. What is the use of seek() method in files?

a) sets the file's current position at the offset
b) sets the file's previous position at the offset
c) sets the file's current position within the file
d) none of the mentioned

EXCEPTION HANDLING

One man's crappy software is another man's full time job. (Jessica Gaston)

EXCEPTION HANDLING

What is an Exception?

- An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

Why use exceptions?

- Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

EXCEPTION HANDLING

How does it work?

- Exception handling enables you handle errors gracefully and do something meaningful about it. Like display a message to user if intended file not found. Python handles exception using try.. except .. block.
- Error handling is done through the use of exceptions that are caught in try blocks and handled in except blocks. If an error is encountered, a try block code execution is stopped and transferred down to the except block.
- The code in the finally block will be executed regardless of whether an exception occurs.

Time to get our hands dirty. Lets try it out

Syntax :-

```
try:
    # write some code
    # that might throw exception
except <ExceptionType>:
    # Exception handler, alert the user
```

EXCEPTION HANDLING

Try and Learn :-

Open a file to just read, file name is “somemissingfile.txt”. Create a Try and Except block to catch the error and print a message like “File doesn’t exist”.

```
try:
    f = open("somemissingfile.txt","r")
    print(f.read())
except IOError:
    print("OOps..file doesnt exist")
```

How did this code work?

1. First statement between try and except block are executed.
2. If no exception occurs then code under except clause will be skipped.
3. If file don’t exist then exception will be raised and the rest of the code in the try block will be skipped
4. When exceptions occurs, if the exception type matches exception name after except keyword, then the code in that except clause is executed.

EXCEPTION HANDLING

What if I had multiple errors to handle?

- What if we were to handle multiple errors from the previous program?
- Suppose, the program is being written by a newbie, and she has a high chance of commiting Syntax errors, as also wrong input?
- Assume , you are an architect for the project, and you have to review the code.
- What would you instruct the developer to do to handle such multiple exceptions?

```
try:
    f = open("foo.txt","r")
    print(f.read())
    f.close()

except IOError:
    print("OOps..file doesnt exist")
except:
    print("Wrong input")
else:
    print("Yeah!! Its a bug free code")
finally:
     print("This will execute no matter what")
```

EXCEPTION HANDLING

Raising Exceptions

- To raise your exceptions from your own methods you need to use raise keyword as shown below

raise ExceptionClass("Your argument")

Try and Learn :

Write a function to input age . If age is less than zero raise an exception that's says that age is less than zero. If age is 0 , print that it's a "New Born". Else check if age is even, or odd and print the same.

```
def ageEvenOrOdd(age):
  if age<0 :
    raise ValueError("Oops , age cant be Less than 0")
  elif age==0:
    print("New Born")
  elif age%2==0:
    print("Age is Even")
  else:
    print("Age is Odd")
```

```
try:
  age = int(input("Enter age"))
  ageEvenOrOdd(age)
except ValueError as e:
  print("Error is",e.args[0])
else:
  print("No problem")
```

EXCEPTION HANDLING

Exception Objects

- An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
   You do your operations here
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

Try and learn :-

Write a program to pass a string argument to a ceil function. Have an exception block to calculate the error, and type the Exact error.

```
import math
try:
    num = int(input("Enter Number"))
    number = math.ceil(num)
except (SyntaxError,TypeError,ValueError) as e:
    print("NameError : Here's the error : \n",e)
```

Stimulants

1. When will the else part of try-except-else be executed?
   a) always
   b) when an exception occurs
   c) when no exception occurs
   d) when an exception occurs in to except block
2. What is the output of the following code?

```
  def foo():
   try:
       return 1
   finally:
       return 2
k = foo()
print(k)
```

a) 1
b) 2
c) 3
d) error, there is more than one return statement in a single try-finally block

3. What is the output of the code shown below?

```
lst = [1, 2, 3]
lst[3]
```

a)NameError
b)ValueError
c)IndexError
d)TypeError

4. An exception is:

a) an object
b) a special function
c) a standard module
d) a module

8. What is the output of below program?

```
def say(message, times = 1):
    print(message * times)
say('Hello')
say('World', 5)
```

a) Hello
WorldWorldWorldWorldWorld
b) Hello
World 5
c) Hello
World,World,World,World,World
d) Hello

9. Where can a function be defined?

a) Module
b) Class
c) Another function
d) All of the mentioned