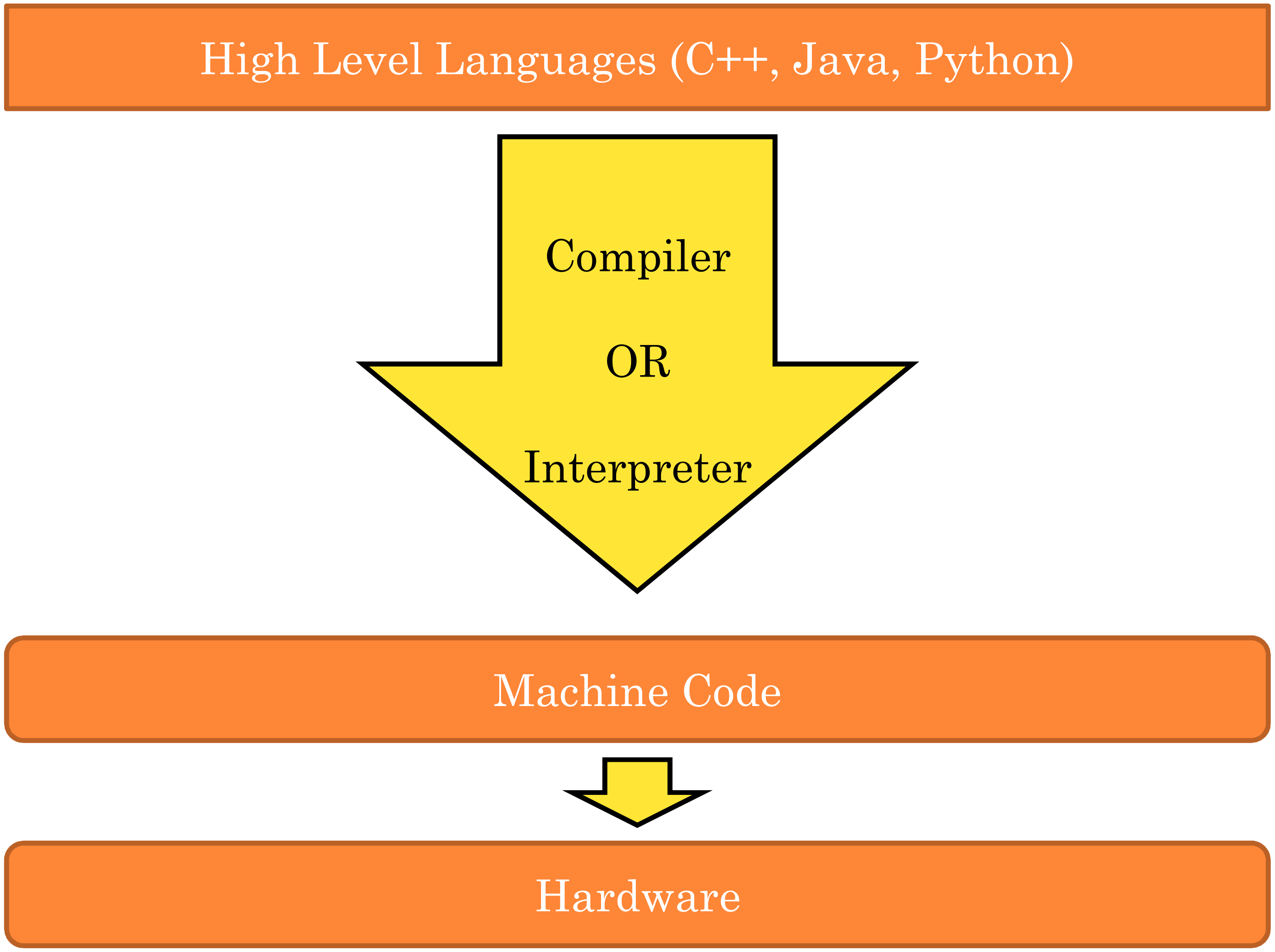
Introduction to Python

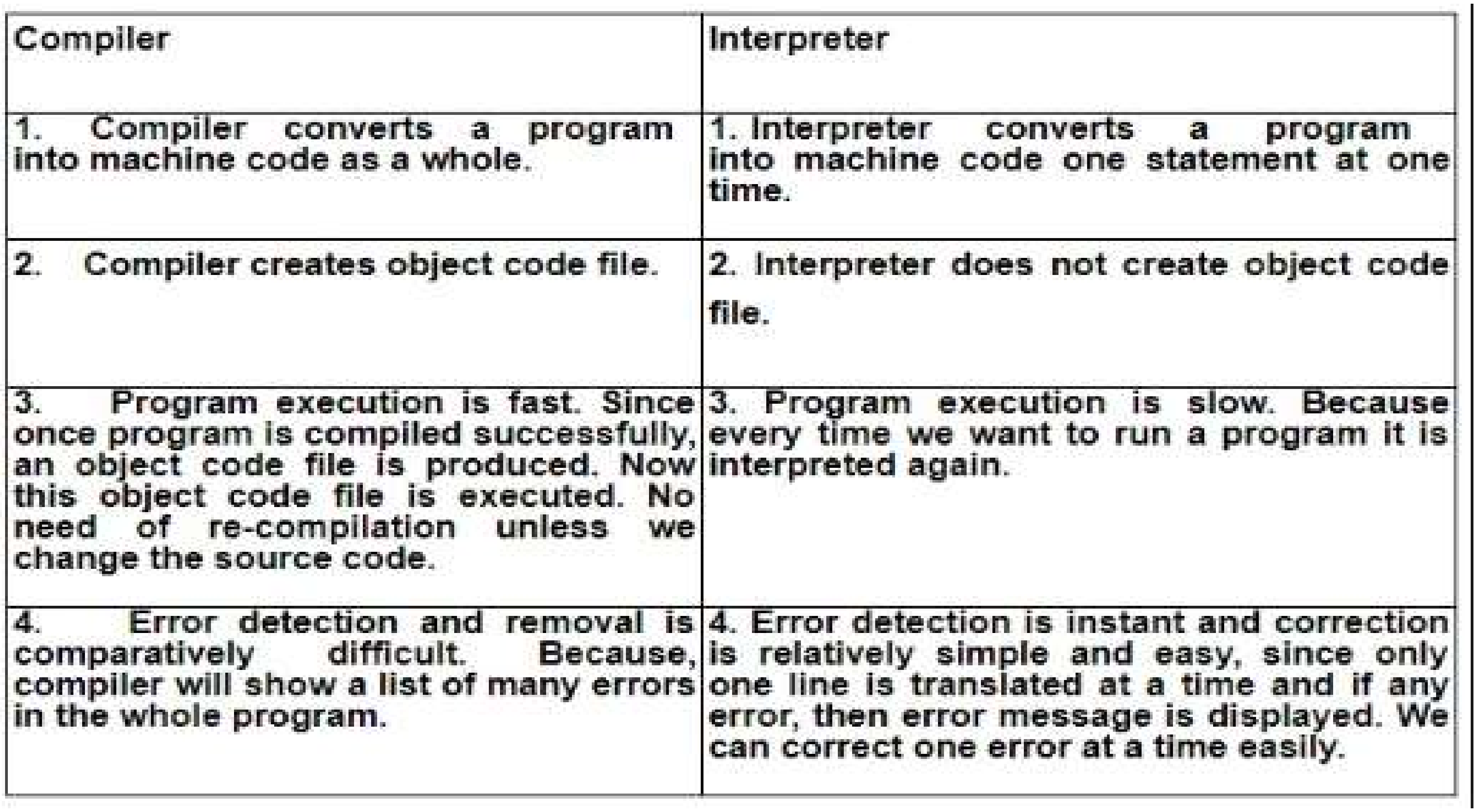
What is python

- Python is a **high-level**, **interpreted**, **interactive and object-oriented scripting language**. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages
- Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

HIGH LEVEL LANGUAGES



DIFFERENCE BETWEEN COMPILER AND INTERPRETER

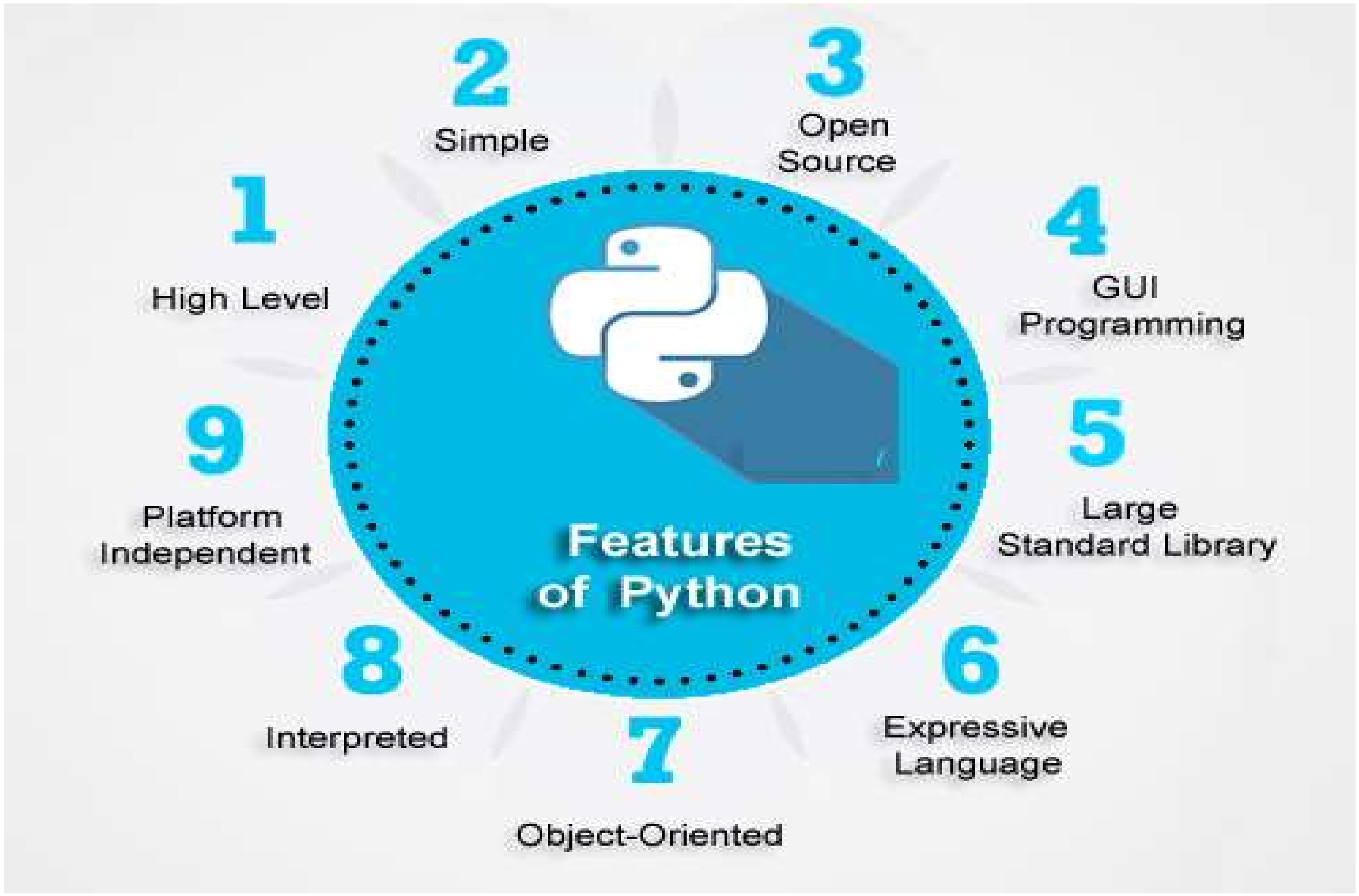


Compiler	Interpreter
1. Compiler converts a program into machine code as a whole.	1. Interpreter converts a program into machine code one statement at one time.
2. Compiler creates object code file.	2. Interpreter does not create object code file.
3. Program execution is fast. Since once program is compiled successfully, an object code file is produced. Now this object code file is executed. No need of re-compilation unless we change the source code.	3. Program execution is slow. Because every time we want to run a program it is interpreted again.
4. Error detection and removal is comparatively difficult. Because, compiler will show a list of many errors in the whole program.	4. Error detection is instant and correction is relatively simple and easy, since only one line is translated at a time and if any error, then error message is displayed. We can correct one error at a time easily.

What is the Python Software Foundation?

- The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF's mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF's home page is at https://www.python.org

Python Features



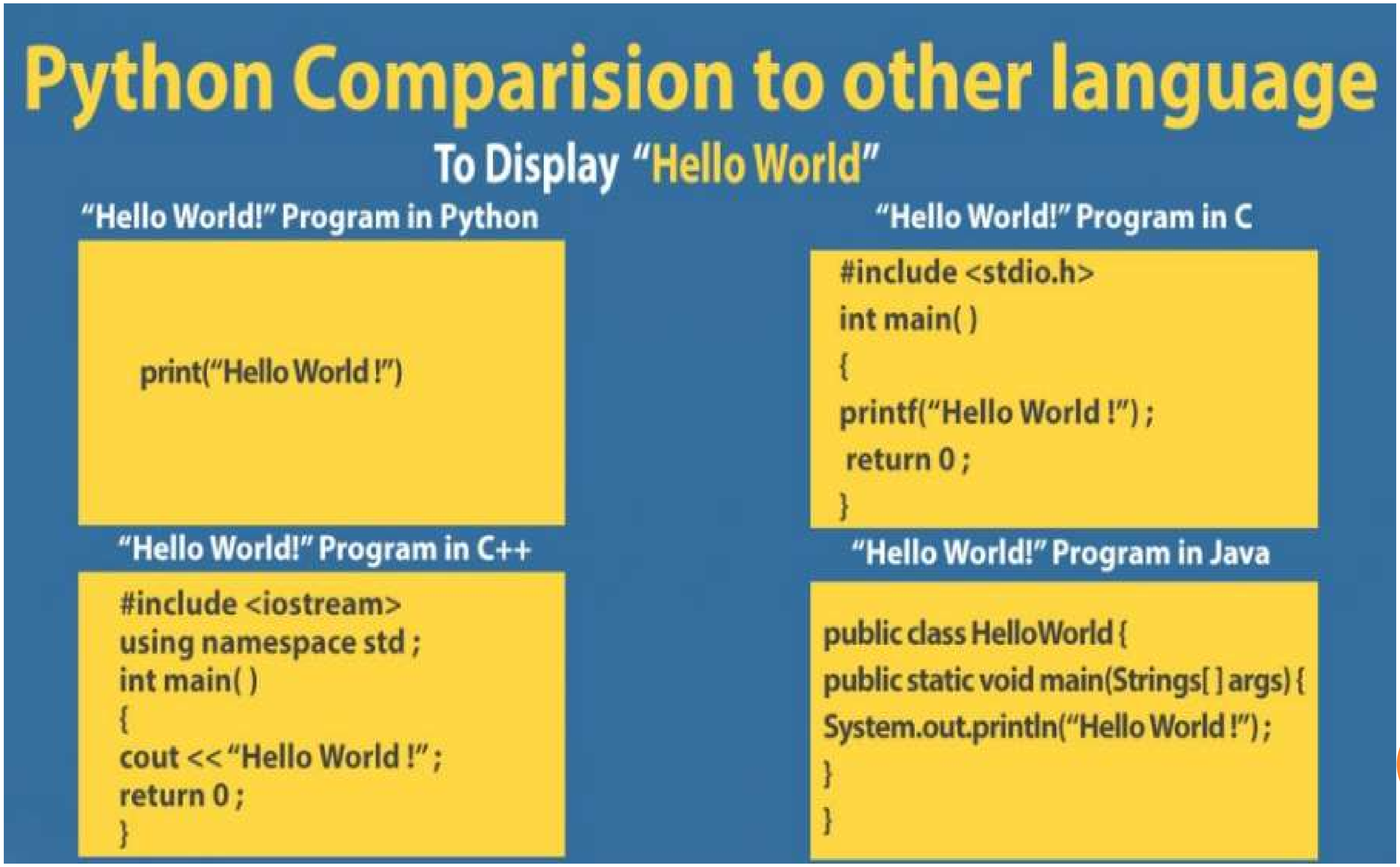
Python Features

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

Python Features

- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs.

Why Python


Python Comparision to other language

To Display "Hello World"

"Hello World!" Program in Python

```
print("Hello World !")
```

"Hello World!" Program in C

```
#include <stdio.h>
int main( )
{
printf("Hello World !") ;
 return 0 ;
}
```

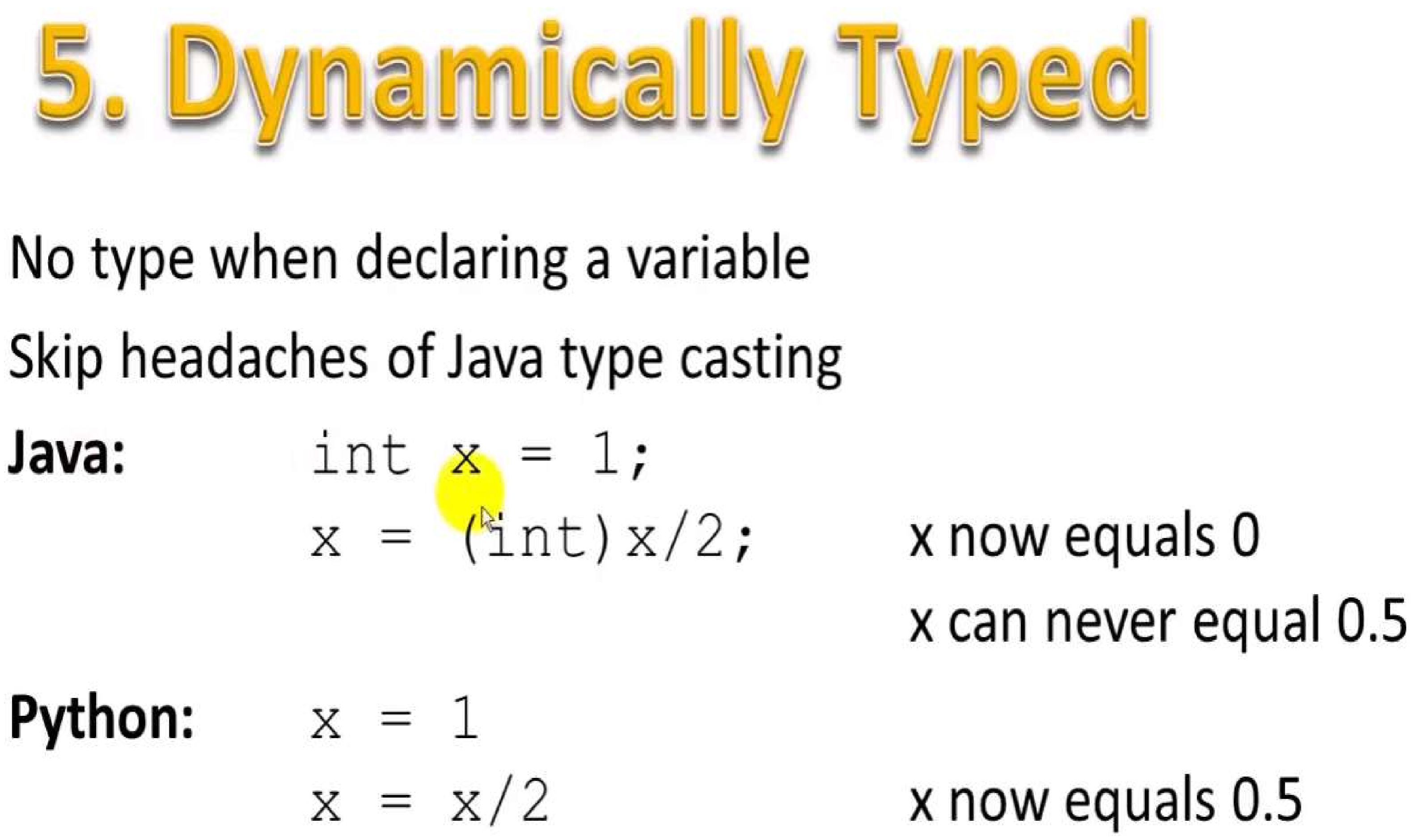
"Hello World!" Program in C++

```
#include <iostream>
using namespace std ;
int main( )
{
cout << "Hello World !" ;
return 0 ;
}
```

"Hello World!" Program in Java

```
public class HelloWorld {
public static void main(Strings[ ] args) {
System.out.println("Hello World !") ;
}
}
```

WHY PYTHON



5. Dynamically Typed

No type when declaring a variable

Skip headaches of Java type casting

Java:

```
int x = 1;
x = (int)x/2;
```

x now equals 0

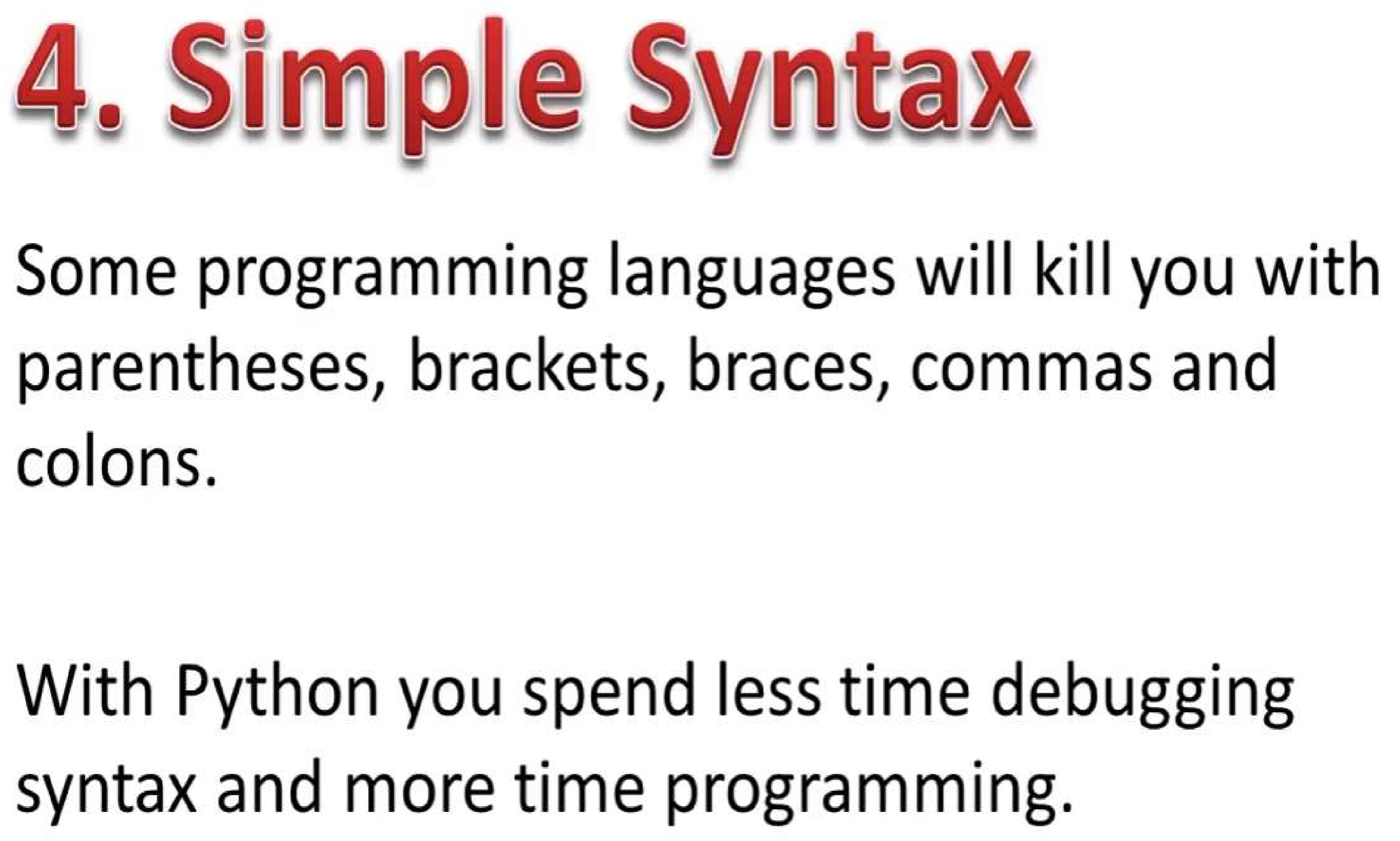
x can never equal 0.5

Python:

```
x = 1
x = x/2
```

x now equals 0.5

Why Python



4. Simple Syntax

Some programming languages will kill you with parentheses, brackets, braces, commas and colons.

With Python you spend less time debugging syntax and more time programming.

Why Python

3. One-Liners

Elegant 1-line solutions to what takes a whole block of code in other languages.

One example: swap x and y

Java:

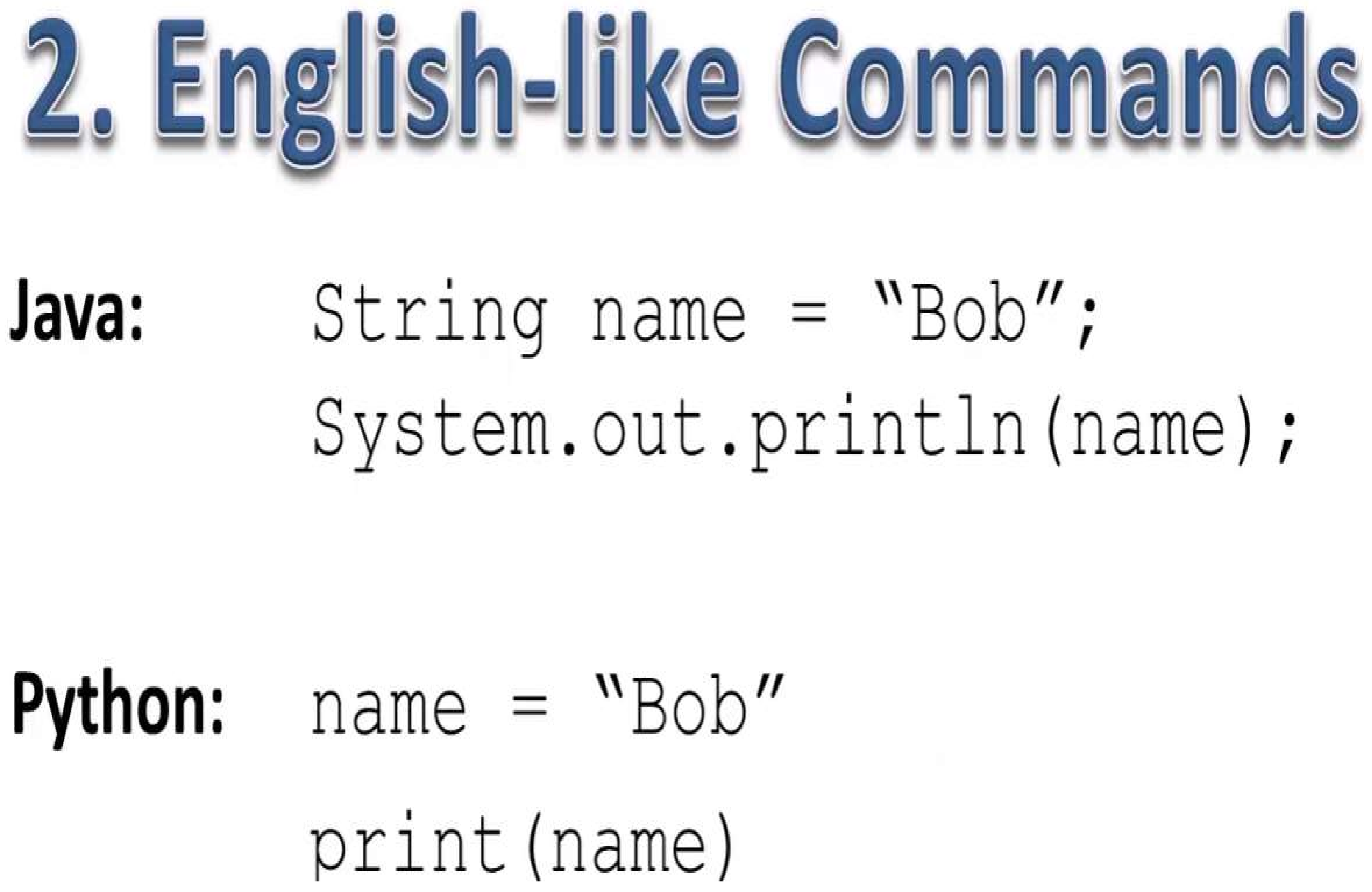
```
int temp = x;
x = y;
y = temp;
```

Python:

```
x,y = y,x
```

Wow! Now that's Pythonic

WHY PYTHON



2. English-like Commands

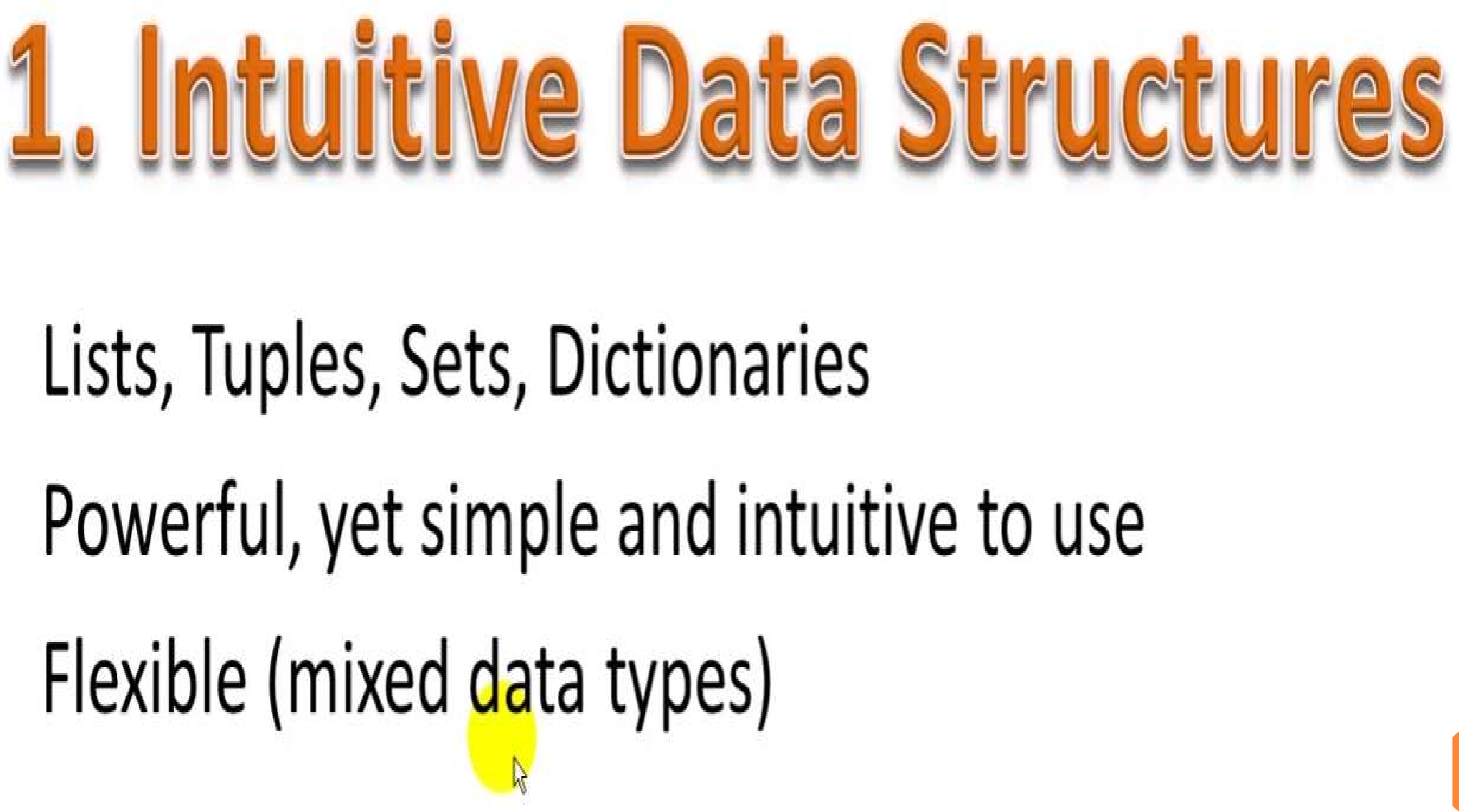
Java:

```
String name = "Bob";
System.out.println(name);
```

Python:

```
name = "Bob"
print(name)
```

Why Python



1. Intuitive Data Structures

Lists, Tuples, Sets, Dictionaries

Powerful, yet simple and intuitive to use

Flexible (mixed data types)

How to write a Python script

- Open editor.

```
#!/usr/bin/python
print "Hello, Python!"
```

- Save the file as .py
- Run the python program

```
 # python Hello.py
```

How to RUN a Python script IN Windows

- python hello.py

What is a shebang line?

It is called a shebang or a "bang" line. It is nothing but the absolute path to the python interpreter. It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /usr/bin/python.

#!path_of_interpreter

Eg: #! /usr/bin/python

Mode Programming

- Interactive Mode Programming

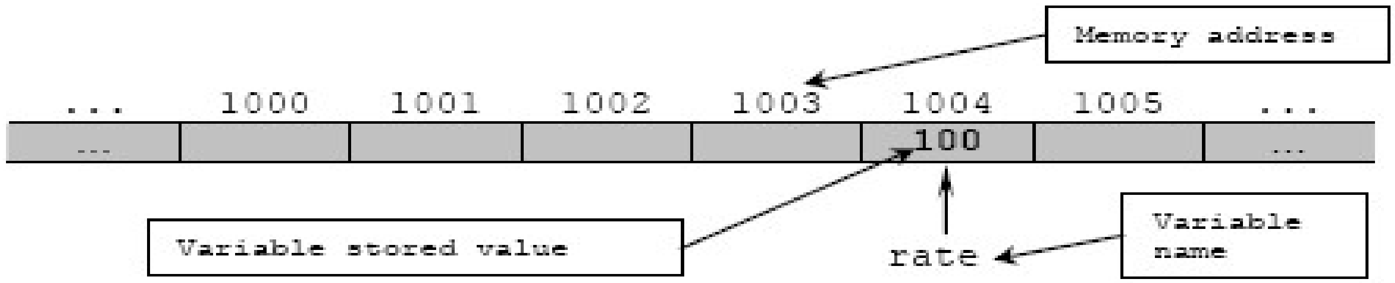
  Invoking the interpreter without passing a script file as a parameter brings up the prompt

- Script Mode Programming

  Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

VARIABLE

- A variable is a object to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. you to create, assign, and delete variables.

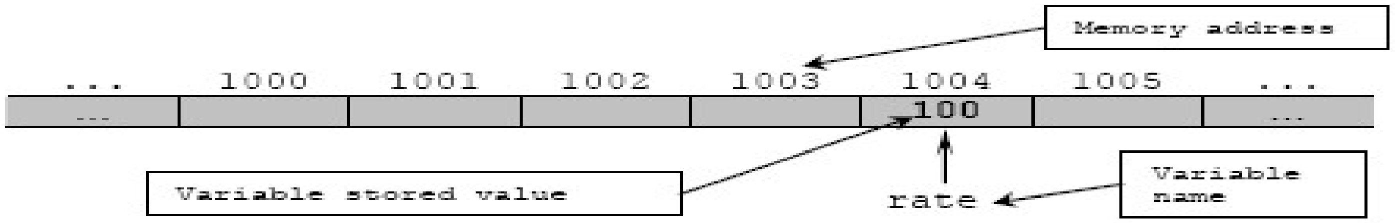


VARIABLE

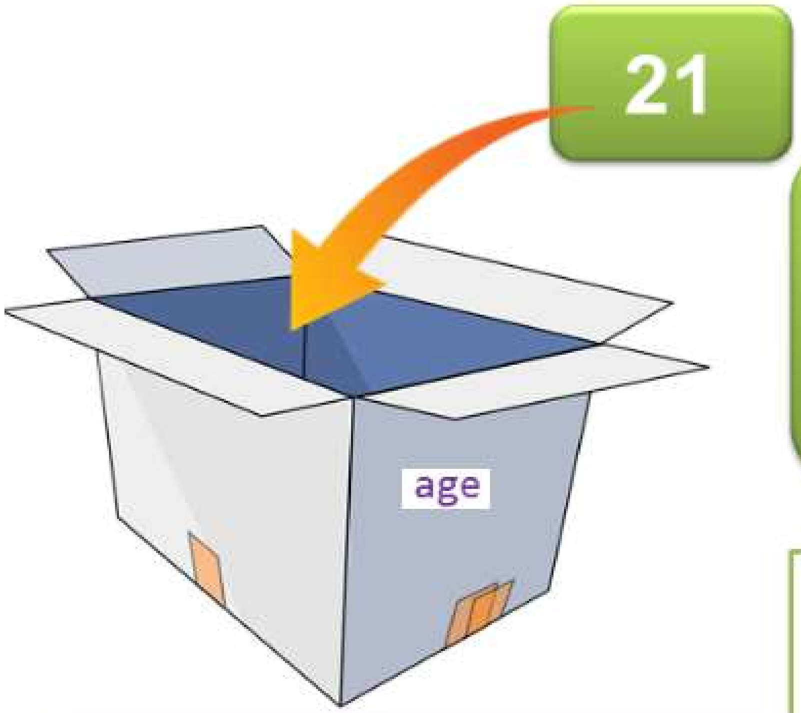
When programming it is often necessary to store a value for use later on in the program.

A variable is a label given to a location in memory containing a value that can be accessed or changed.

Think of a variable as a box with a label that you can store information in.



VARIABLE



We can think that variable is one type of Container where we can store some element

```
int = which type of element we
         can store
age = name of the container box

   21 = type of element
```

```
age = 21
```

Rules For Constructing Variable / Function /Class Name (Identifiers)

- **Characters Allowed :**
  - Underscore(_)
  - Capital Letters (A – Z)
  - Small Letters (a – z)
  - Digits (0 – 9)
- **Blanks & Commas** are not allowed
- No Special Symbols other than **underscore(_) are allowed**
- **First Character** should be **alphabet or Underscore**
- Variable name Should not be **Reserved Word**

Reserved Words

And	exec	Not
Assert	finally	or
Break	for	pass
Class	from	print
Continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Rules For Constructing Variable Name

- **Valid Names**

num
Num
Num_1
NUM_temp2
_temp
Days_365

- **Invalid Names**

number 1

num,1
addition of program
1num
1_num
365_days

Variable creation and Assigning Value

- Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable

```
Name = "Anil"
Age = 20
A = 100.10
```

Accessing Variable

```
name = “Anil”
age = 20

print name
print age
```

PRINT

- Print is function to print the output on STD terminal.
- By default in python, print will print the output on new line.

examples :

- print ("Hello, Python!")
- name= “Anil”

  print (“Name : " , name)

  print (“Name : %s" %name)
- name= “Anil”

  last_name= “Ambani ”

  print (“Name : %s %s" %(name, last_name))
- age = 20

  print (“age : %d" % age)

  print (“age : %d" , age)

COMMENTS

- # is used to comment the line.
- This is helpful to add the comments in script.
- Commented line will not get executed.

Example :

$vi hello_world.py

```
# This will print "Hello, World"
print "Hello, world"
```

INPUT

- Let's have a look at the following example:

```
name= input("What's your name? ")
print("Nice to meet you " , name)
```

Types of Operator

- Python language supports the following types of operators.
  - Arithmetic Operators
  - Comparison (Relational) Operators
  - Assignment Operators
  - Logical Operators
  - Bitwise Operators
  - Membership Operators
  - Identity Operators

Python Arithmetic Operators

Operator	Description	Example (a=10 b=20)
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

Operator	Description	Example (a=10 b=20)
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) true
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Operator	Description	Example (a=10 b=20)
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Standard Data Types

Python has five standard data types

- Numbers
- String
- List
- Tuple
- Dictionary

Strings

STRING

- A string is simply a series of characters. Anything **inside quotes** is considered a string in Python, and you can use single or double quotes around your strings like this:

  "This is a string"

  'This is also a string.'

- This flexibility allow you to use quotes and apostrophes within your strings.

'I told my friend, "Python is my favorite language!" '

"It's Done"

String Methods

- **len(string)**
  Returns the length of the string

- Example :

```
str = "this is string example....wow!!!";
print "Length of the string: ", len(str)
```

STRING METHODS

- .upper() & .lower()

  The .upper() and .lower() string methods are self-explanatory. Performing the .upper() method on a string converts all of the characters to uppercase, whereas the lower() method converts all of the characters to lowercase.

  Example :

  Str1 = “example string lower to uper”

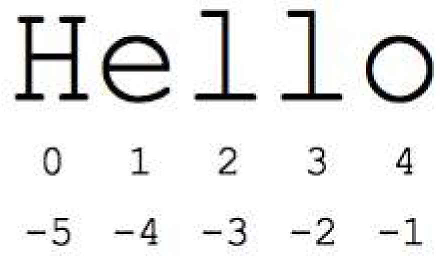
  Str.upper()

  Str2 = “ABC IS COMPANY”

  Str2.lower()

STRING SLICES

- Use [start : end] to get set of letter
- Keep in mind that python, as many other languages, starts to count from 0!!



word = "Hello World"

- print word[0]   #get one char of the word
- print word[0:1]   #get one char of the word (same as above)
- print word[0:3]   #get the first three char
- print word[:3]   #get the first three char
- print word[-3:]   #get the last three char
- print word[3:]   #get all but the three first char
- print word[:-3]   #get all but the three last character

String Special Operators

operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[ ]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

String Methods

- .count()

The .count() method adds up the number of times a character or sequence of characters appears in a string. For example:

- Example :

str = "That that that is not is not is that it it is“

str.count("t")

str.count("that")

String Methods

- .find()

We search for a specific character or characters in a string with the .find() method.

- Example

str = "On the other hand, you have different fingers."

str. find("hand")

String Methods

- .replace()

Let’s say we want to increase the value of a statement. We do so with the .replace() method.

- Example

```
>>> str = "I intend to live forever, or die trying."
>>> str.replace("to", "three")
'I intend three live forever, or die trying.'
>>> str.replace("fore", "five")
'I intend to live fivever, or die trying.'
```

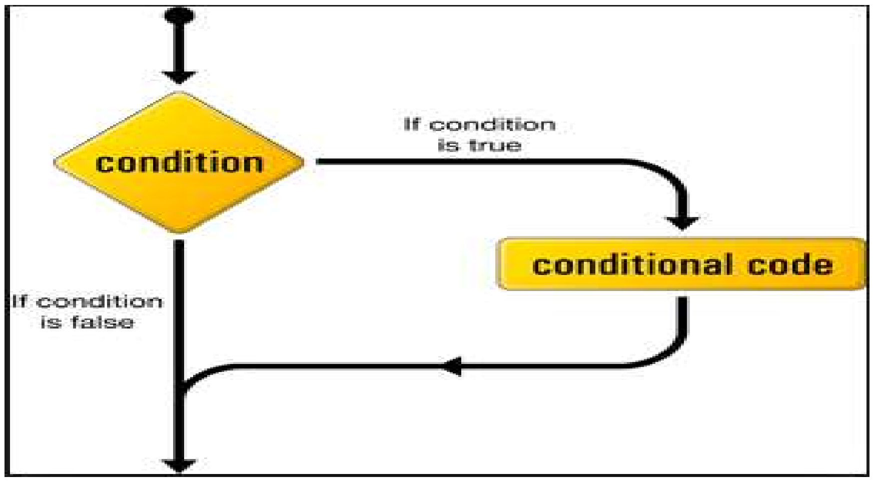
Decision Making

DECISION MAKING

- if statements
- if...else statements
- if...elif...else statements
- nested if statements

IF STATEMENT

- If the boolean expression evaluates to **true** then the block of code inside the if statement will be executed. If boolean expression evaluates to **false** then the first set of code after the end of the if statement(after the closing curly brace) will be executed.



IF STATEMENTS

- The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

- Syntax

```
if expression:
    statement(s)
    statement(s)
print “End of sciprt”
```

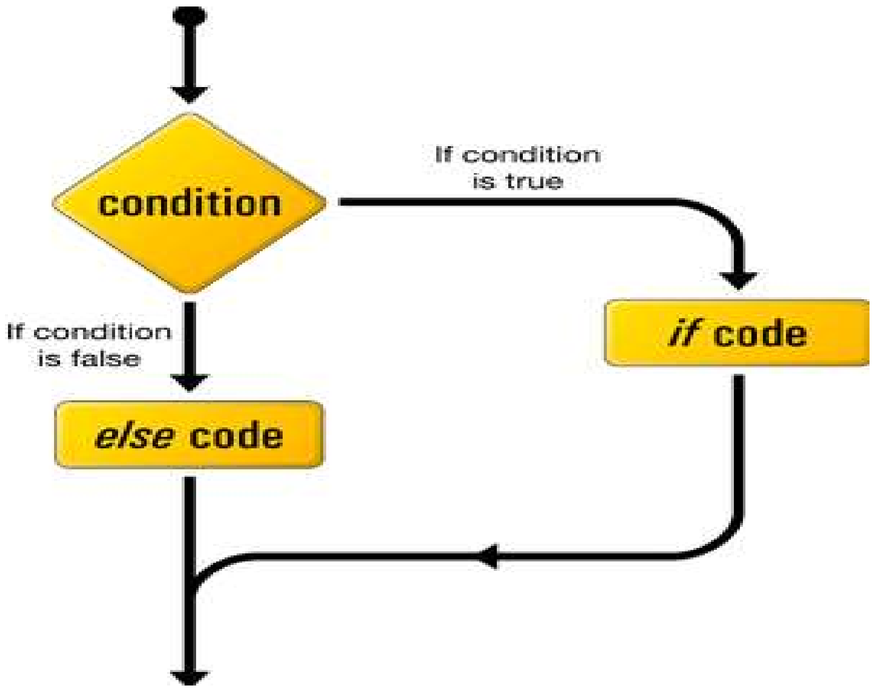
IF…ELSE…FI STATEMENT

- The **if...else...** statement is the next form of control statement that allows user to execute statements in more controlled way and making decision between two choices.
- Syntax

```
if expression:
        statement(s)
else:
        statement(s)
```

IF…ELSE…FI STATEMENT

If the boolean expression evaluates to **true,** then the **if block** of code will be executed otherwise **else block** of code will be executed.

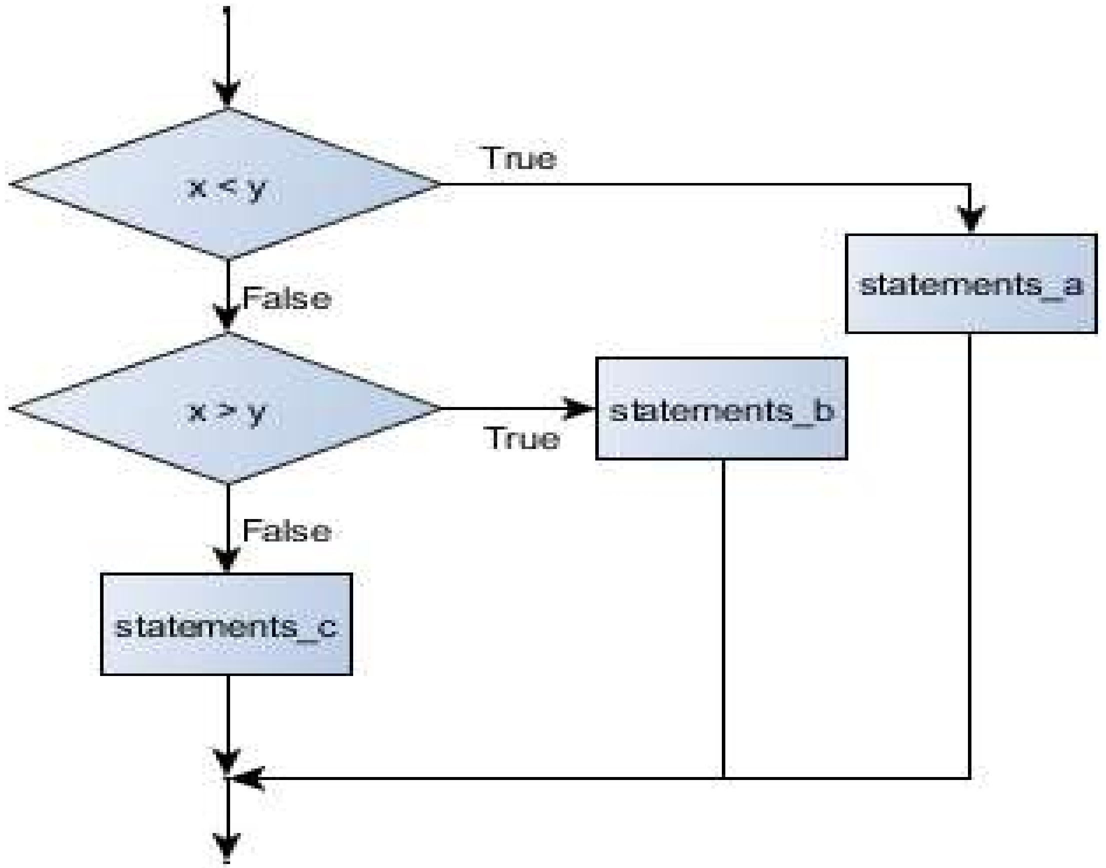


IF…ELIF…FI STATEMENT

- **if...elif...** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.
- Syntax

```
if [ expression 1 ] :
        Statement(s) to be executed if expression 1 is true
elif [ expression 2 ] :
        Statement(s) to be executed if expression 2 is true
elif [ expression 3 ] :
        Statement(s) to be executed if expression 3 is true
else :
        Statement(s) to be executed if no expression is true
```

IF...ELIF...FI STATEMENT



NESTED IF STATEMENTS

- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if**construct.
- In a nested **if** construct, you can have an **if...elif...else** construct inside another**if...elif...else** construct.

NESTED IF STATEMENTS

Syntax :

```
if expression1:
        statement(s)
       if expression2:
                    statement(s)
       elif expression3:
                    statement(s)
      else
                   statement(s)
elif expression4:
                   statement(s)
else:
                   statement(s)
```

SINGLE STATEMENT SUITES

- If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.
- **one-line if** clause
- Example :

```
var = 100
if ( var == 100 ) : print "Value of expression is 100"
print "Good bye!"
```

Loops

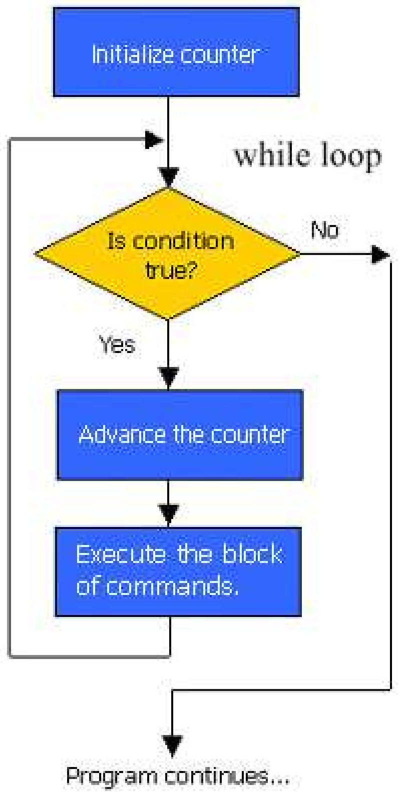
Loops

- A loop statement allows us to execute a statement or group of statements multiple times.
    - while loop
    - For loop

WHILE LOOP

- The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.
- Syntax :

```
while condition :
        Statement(s)
```

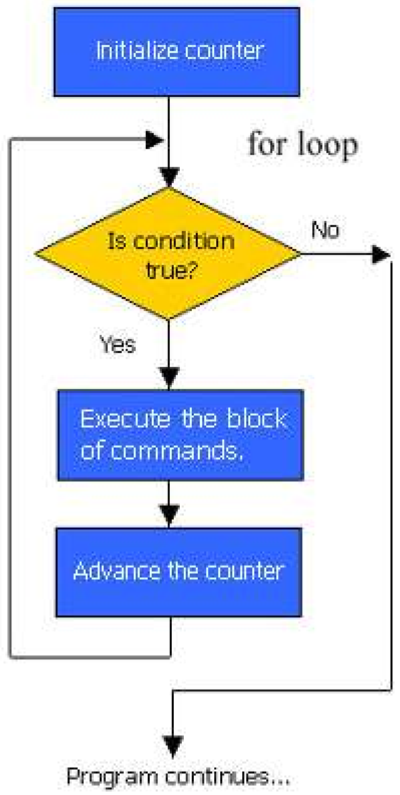


For loop

- The for loop operate on lists of items. It repeats a set of commands for every item in a list.

- Method 1 Syntax :

```
for var in word1 word2 ... wordN
        command1
        command2
        ..
```



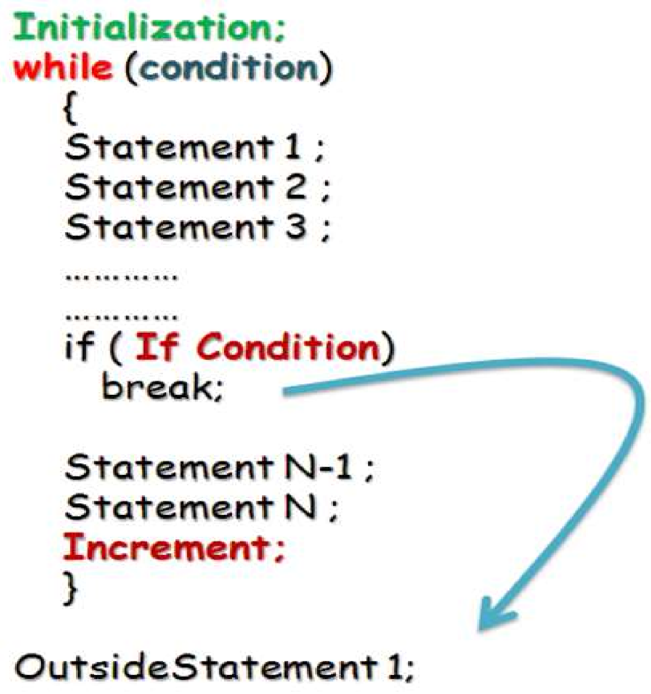
Loop Control Statements

- break statement
- continue statement

Break

- The most common use for break is when some external condition is triggered requiring a hasty **exit from a loop**. The **break** statement can be used in both *while* and *for* loops.
- Syntax

  break

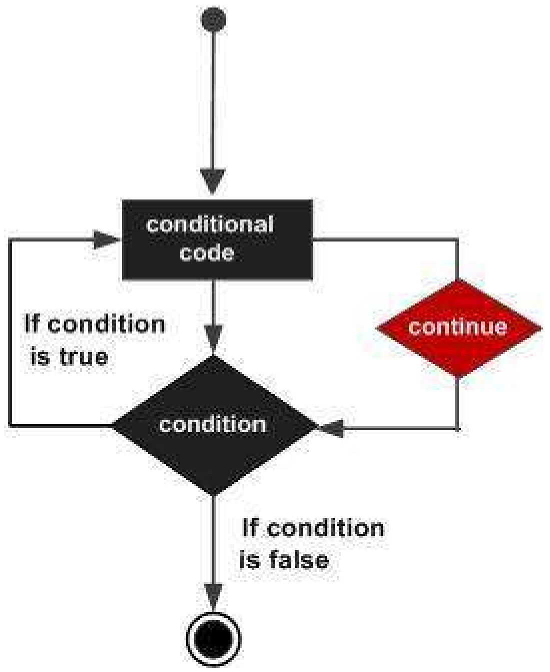


```
Initialization;
while (condition)
    {
    Statement 1 ;
    Statement 2 ;
    Statement 3 ;
    ............
    ............
    if ( If Condition)
      break;

    Statement N-1 ;
    Statement N ;
    Increment;
    }

OutsideStatement 1;
```

CONTINUE

- It returns the control to the beginning of the while loop.. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- Syntax

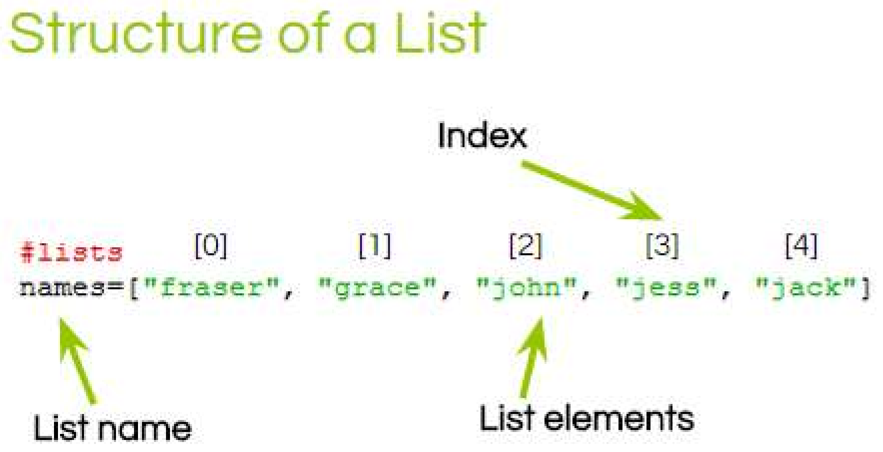
  contiue



LIST

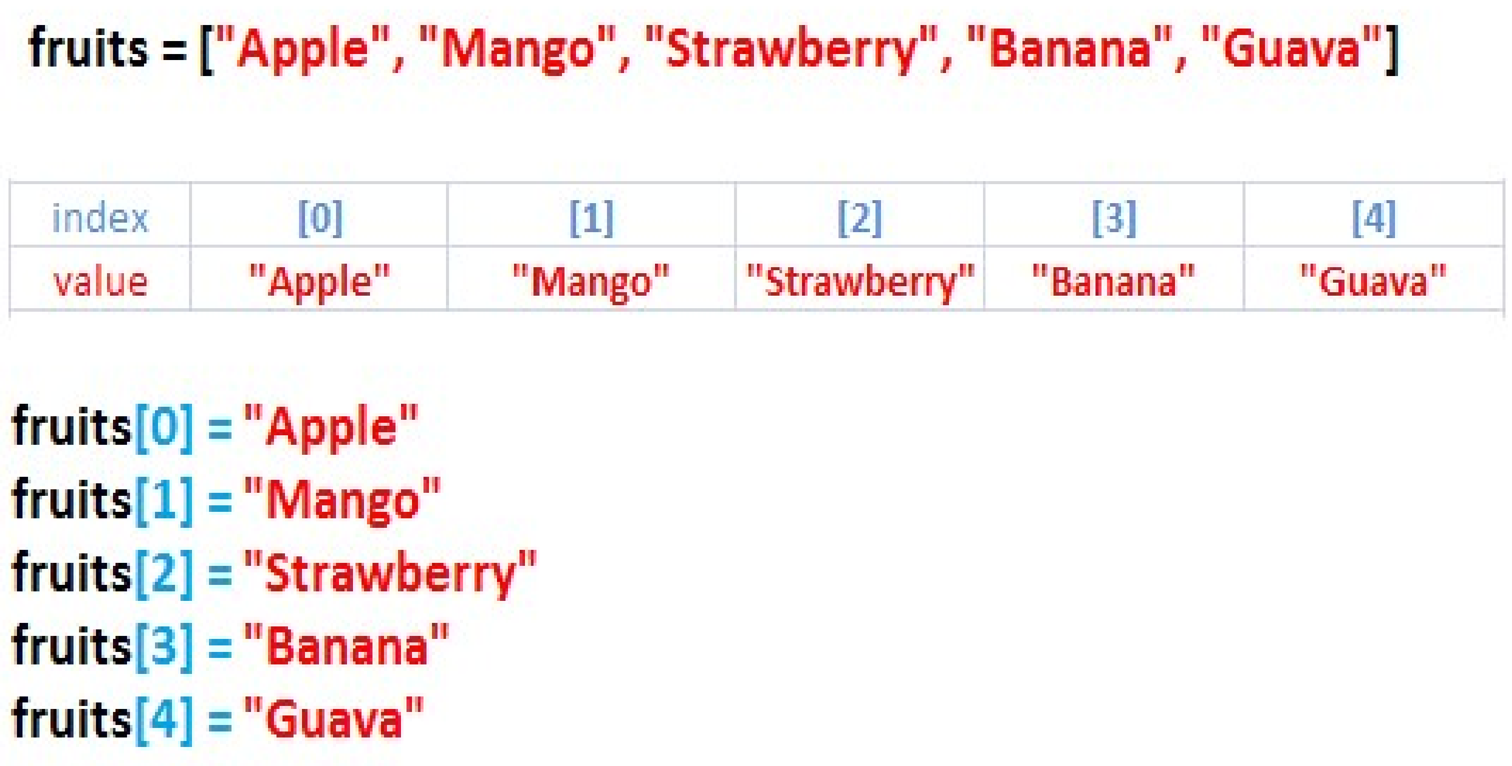
LISTS

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets ([ ]).
- To some extent, lists are similar to arrays in C.
- One difference between them is that all the items belonging to a list can be of different data type.



```
#lists
names=["fraser", "grace", "john", "jess", "jack"]
```

LIST



```
fruits = ["Apple", "Mango", "Strawberry", "Banana", "Guava"]
```

index	[0]	[1]	[2]	[3]	[4]
value	"Apple"	"Mango"	"Strawberry"	"Banana"	"Guava"

```
fruits[0] = "Apple"
fruits[1] = "Mango"
fruits[2] = "Strawberry"
fruits[3] = "Banana"
fruits[4] = "Guava"
```

ACCESSING LIST

- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.

- list1 = [‘anil’, 10 , 5.23, “Sunil”, 88.2]

```
print list1          # Prints complete list print
print list1[0]       # Prints first element of the list
print list1[1:3]     # Prints elements starting from 2nd till 3rd
print list1[2:]      # Prints elements starting from 3rd element
```

Change value in list

- The values stored in a list can be changed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.

```
names[0] = "Nilesh"
names[-1] = "Anana"
```

ADDING ELEMENT TO A LIST

- The simplest way to add a new element to list is to **append** the item to the list.

- Example :

```
motorcycles = [‘honda’, ’yamaha’, ‘suzuki’]
motorcycle.append(‘tvs’)
print(motorcycles)
```

- Output :

```
[‘honda’, ’yamaha’, ‘suzuki’,’tvs’]
```

Inserting Elements into a list

- You can add anew element at any position in your list by using the **insert()** method.
- Example :

```
motorcycles = [‘honda’, ’yamaha’, ‘suzuki’]
motorcycle.insert(0,‘tvs’)
print(motorcycles)
```

- Output :

```
[‘tvs’,‘honda’, ’yamaha’, ‘suzuki’]
```

Delete List Elements

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
subject = ['physics', 'chemistry', ‘math’, ’Biology’ ]
print subject
del subject[2]
print "After deleting value at index 2 : "
print subject
```

REMOVING AN ITEM BY VALUE

- Sometimes you won’t know the position of the value you want to remove from a list.
- If you only know the value of the item you want to remove, you can use **remove()** method.
- Example :

```
motorcycles = [‘honda’, ’yamaha’, ‘suzuki’]
motorcycle.remove(’yamaha’)
print(motorcycles)
```

- Output :

```
[‘honda’, ‘suzuki’]
```

LIST CONCATENATION

- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.
- Examples:

```
alist = [ 'anil', 10 , 5, "Sunil", 88 ]
blist = [123, 'kumar']

print blist * 2      # Prints list two times
print alist + btinylist # Prints concatenated lists
```

Basic List Operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
List1=[10,20,30] 20 in List1	True	Membership
for x in List1: print x,	1 2 3	Iteration

Built-in List Functions & Methods:

Function	Description
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.
min(list)	Returns item from the list with min value.
list(tupe)	Converts a tuple into list.
sorted	Take elements in the list and return a new sorted list
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

Tuples

TUPLE

- A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the **tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.**
- Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

TUPLES

- Syntax :

```
tuple_name = (val1, val2, ...)
```

- Example :

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
```

Accessing Values in Tuples

- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.

- Example :

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
print "tup1[0]: ", tup1[0]
print "tup2[1:5]: " tup2[1:5]
```

Updating Tuples

- Tuples are immutable which means **<u>you cannot update or change the values of tuple elements.</u>**

Delete Tuple Elements

- **Removing individual tuple elements is not possible**. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

- To explicitly remove an entire tuple, just use the **del** statement.

Basic Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Tuple Functions

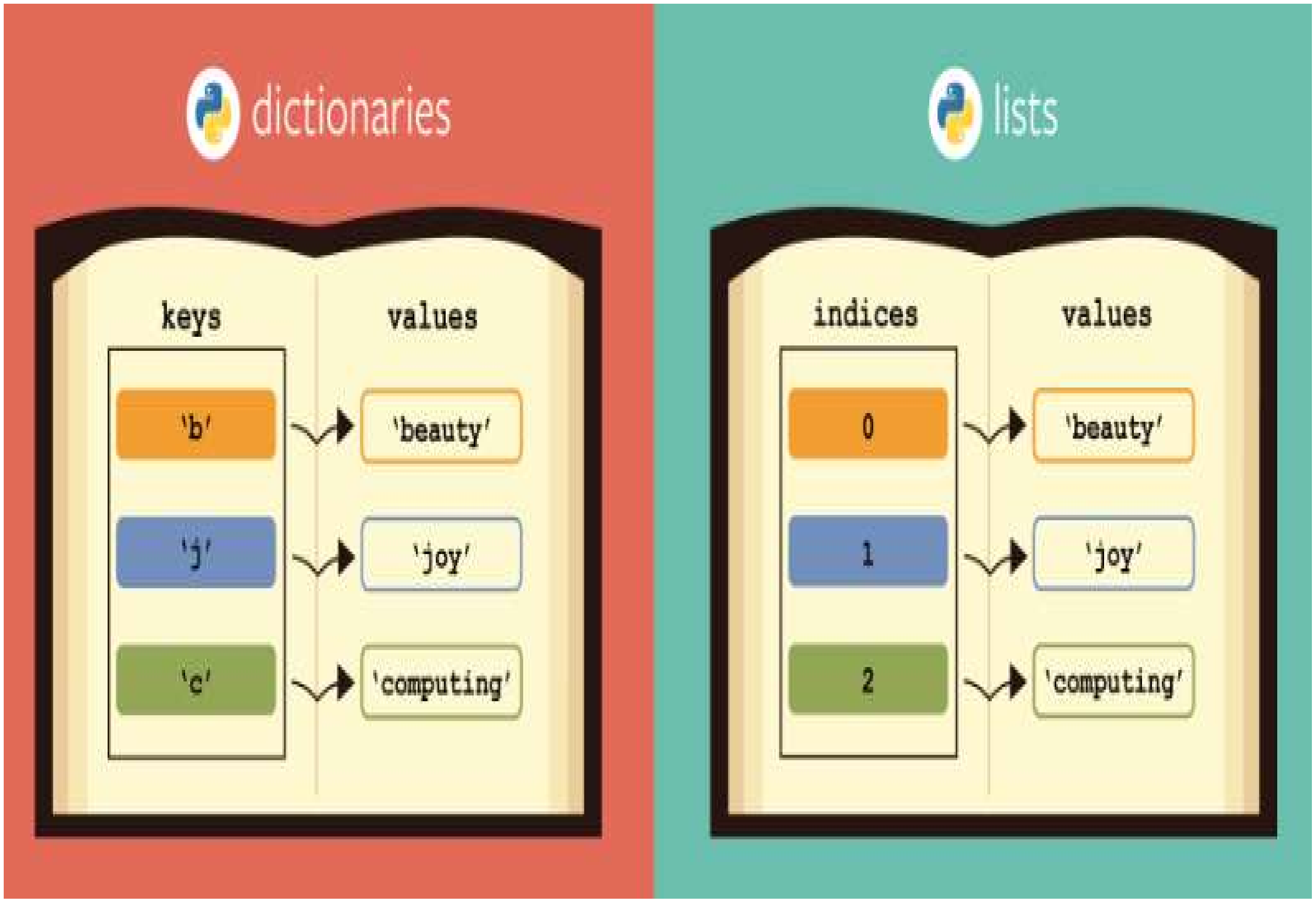
Function	Description
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns item from the tuple with max value.
min(tuple)	Returns item from the tuple with min value.
tuple(seq)	Converts a list into tuple

Dictionary

Dictionaries

- **Dictionaries** are yet another kind of compound type. They are Python's built-in **mapping type**. They map **keys**, which can be any immutable type, to values, which can be any type (heterogeneous), just like the elements of a list or tuple. In other languages, they are called associative arrays since they associate a key with a value.

Dictionaries



DICTIONARIES

- Each key is separated from its value by a colon (:)
- Items are separated by commas.
- Whole thing is enclosed in curly braces.
- Keys are unique within a dictionary while values may not be.
- The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.
- Syntax :

  Dictionaries = { key : value, key : value, ... }

ACCESSING VALUES IN DICTIONARY:

Syntax :

```
            Dictionary[key]
```

Example :

```
states = { 'Oregon': 'OR',
              'Florida': 'FL',
             'California': 'CA',
             'New York': 'NY',
              'Michigan': 'MI‘
            }

print "NY State has: ", states['NY']
```

Adding and changing values

```
capitals = {
            'Maharashtra' : 'Mumbai',
            'Telangana' : 'Hyderabad',
            'Andhra Pradesh' : 'Hyderabad',
            'Delhi' : 'Delhi',
            'Madhya Pradesh' : 'Bhopal'
           }

print "Before : ", capitals

# Adding
capitals['Goa'] = 'Panaji'
capitals['Gujarat'] = 'Gandhinagar'
capitals['Karnataka'] = 'Bangalore'
print "After :", capitals

# updating / Changing some more cities
capitals['Andhra Pradesh'] = 'Amravathi'
print "Capital of AP will be ", capitals['Andhra Pradesh']
```

Delete Dictionary Elements

- del states['DL']   # remove entry with key 'Name'
- dict.clear()   # remove all entries in dict
- del dict   # delete entire dictionary

Dictionary and functions

Function	Description
dict.items()	Returns a list of *dict*'s (key, value) tuple pairs
dict.values()	Returns list of dictionary *dict*'s values
dict.keys()	Returns list of dictionary dict's keys
dict.get(key, default=None)	For *key* key, returns value or default if key not in dictionary
dict.setdefault(key, default=None)	Similar to get(), but will set dict[key]=default if *key* is not already in dict
dict.has_key(key)	Returns *true* if key in dictionary *dict*, *false* otherwise
dict.clear()	Removes all elements of dictionary *dict*
dict.copy()	Returns a shallow copy of dictionary *dict*
dict.fromkeys()	Create a new dictionary with keys from seq and values *set* to *value*.

Dictionary and functions

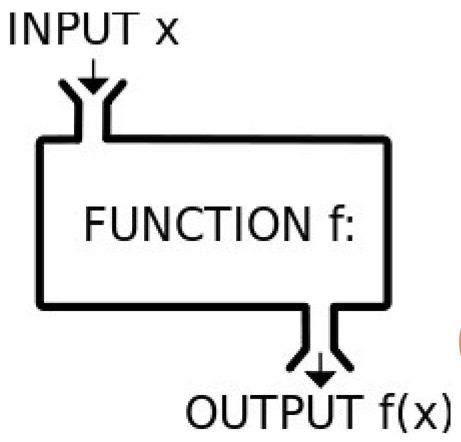
Function	Description
len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

Function

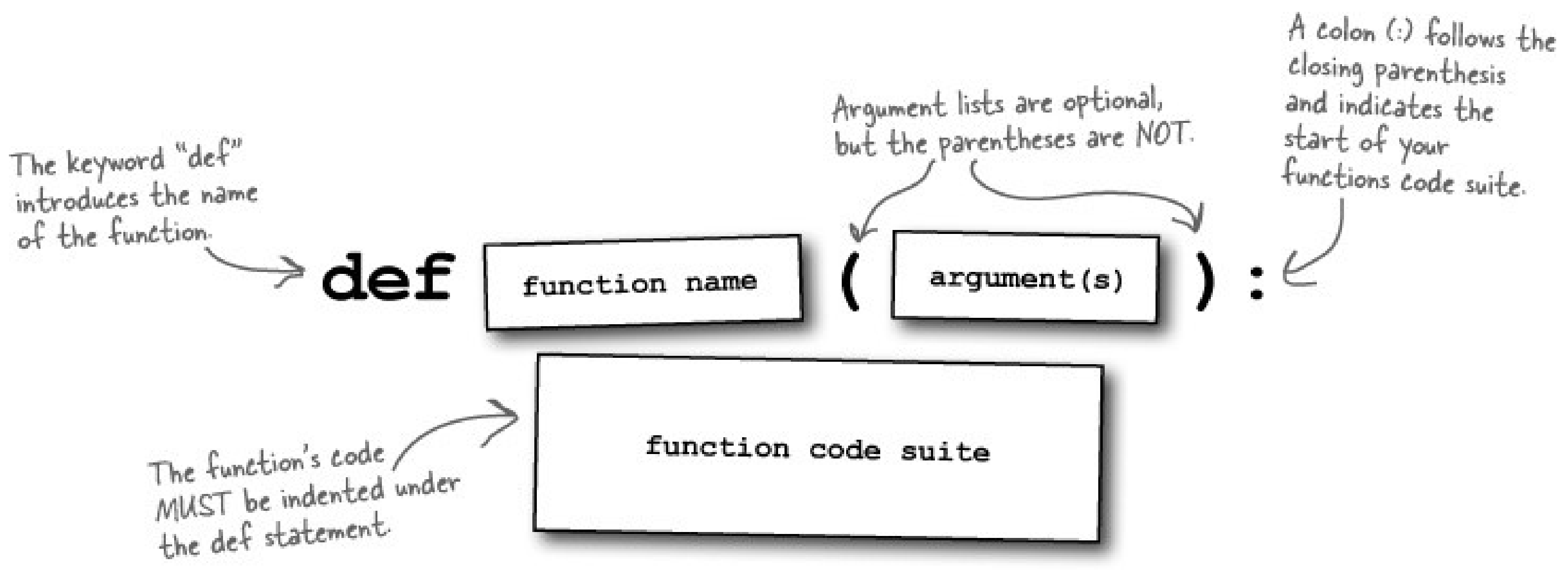
FUNCTIONS

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. You have already seen various functions. These are called built-in functions provided by the language itself, but we can write our own functions.
- Function can take inputs and

Returns the value



Defining a Function


The keyword "def" introduces the name of the function.
def
function name
(
argument(s)
)
:
Argument lists are optional, but the parentheses are NOT.
A colon (:) follows the closing parenthesis and indicates the start of your functions code suite.
function code suite
The function's code MUST be indented under the def statement.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( ))
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

DEFINING A FUNCTION

- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.
- Syntax :

```
def functionname( parameters ):
        "function docstring“
        function_suite
        return [expression]
```

Function Scope

- Python does support global variables without you having to explicitly express that they are global variables. It's much easier just to show rather than explain

```
def someFunction():
        a = 10
someFunction()
print a
```

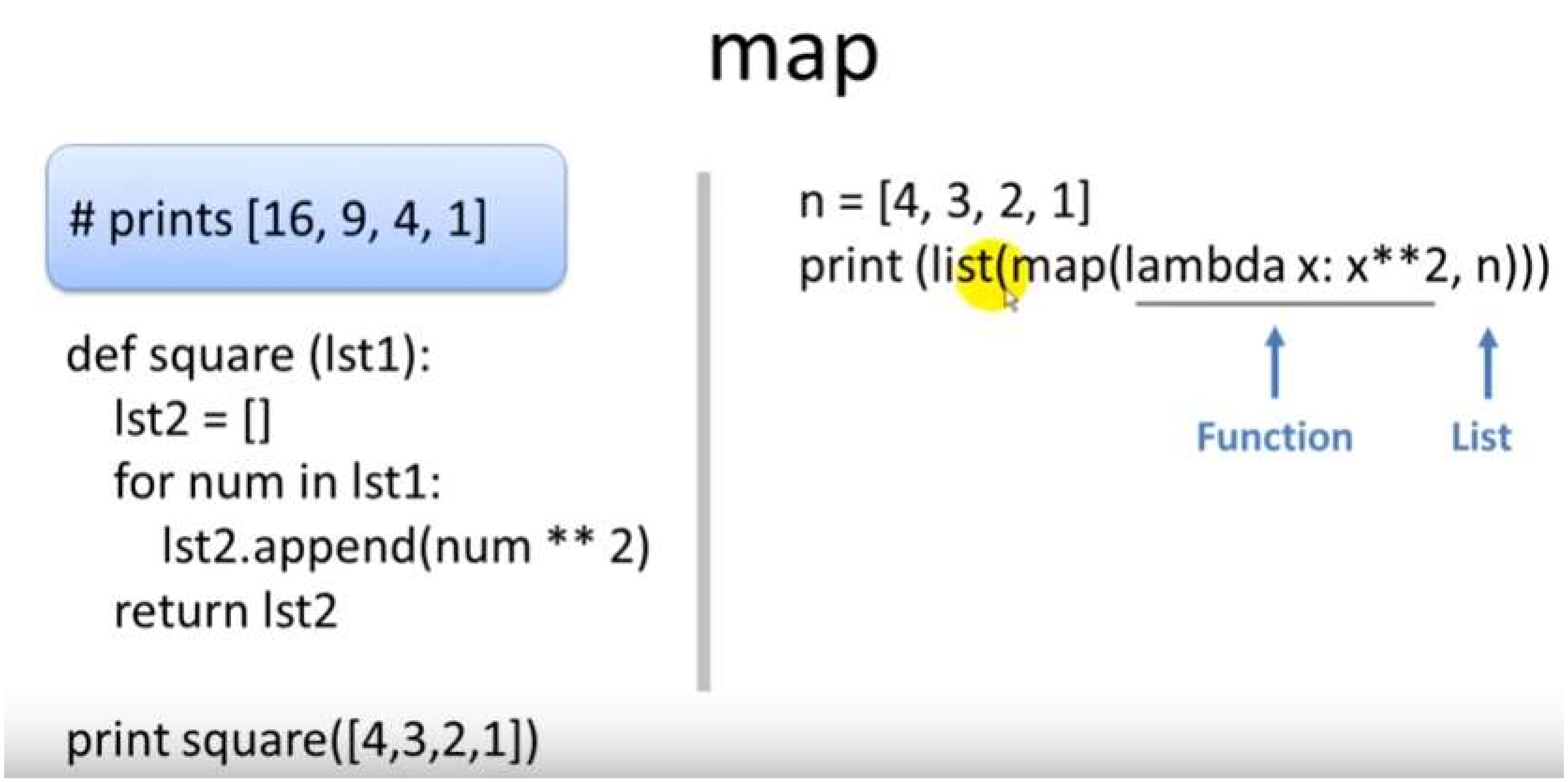
- This will cause an error because our variable, a, is in the local scope of **someFunction**. So, when we try to print a, Python will bite and say a isn't defined. Technically, it is defined, but just not in the global scope.

```
a = 10
def someFunction():
        print a

someFunction()
```

Lambda

- These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword. You can use the **lambda** keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

LAMBDA MAP

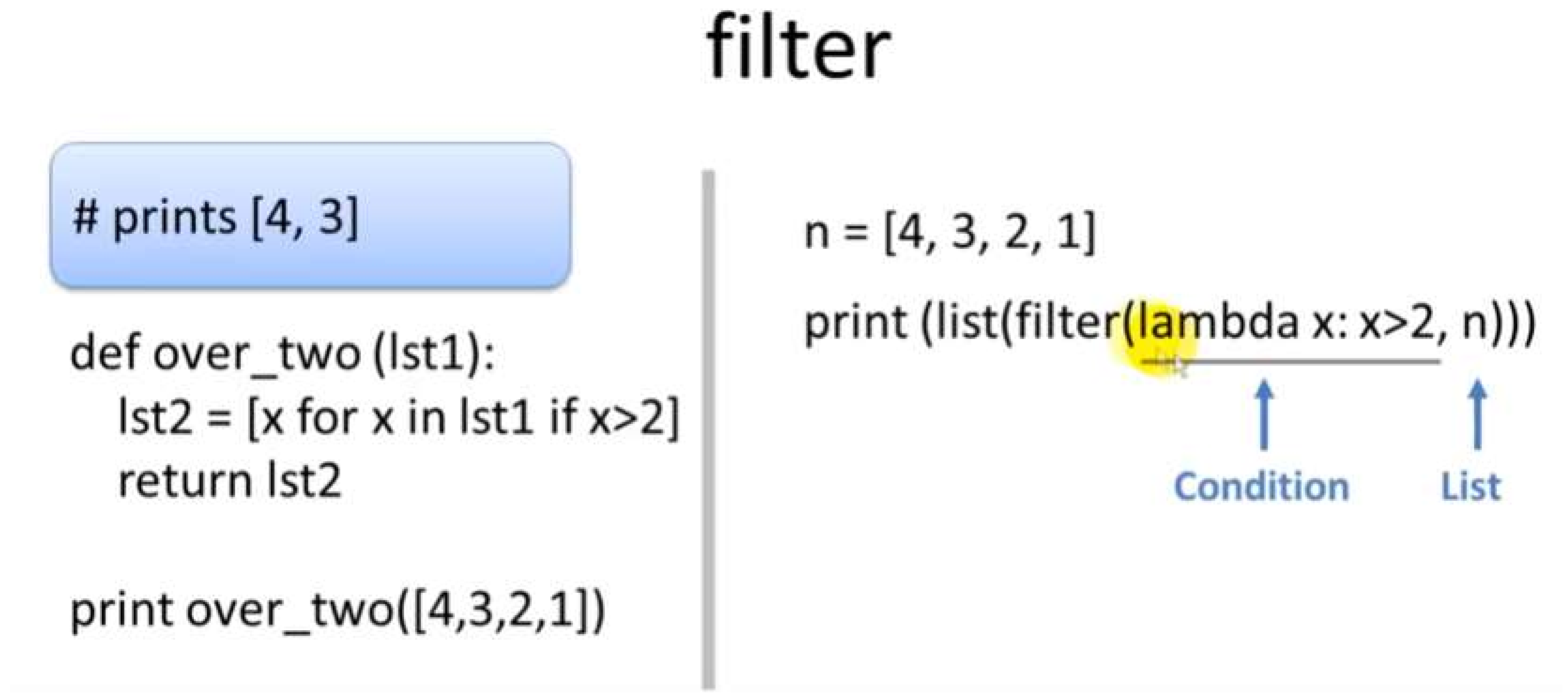


map

```
# prints [16, 9, 4, 1]

def square (lst1):
   lst2 = []
   for num in lst1:
      lst2.append(num ** 2)
   return lst2

print square([4,3,2,1])
```

```
n = [4, 3, 2, 1]
print (list(map(lambda x: x**2, n)))
```

Lambda Filter


filter

prints [4, 3]

```
def over_two (lst1):
    lst2 = [x for x in lst1 if x>2]
    return lst2

print over_two([4,3,2,1])
```

```
n = [4, 3, 2, 1]

print (list(filter(lambda x: x>2, n)))
```

Modules

Modules

- Modules in Python are simply Python files with the .py extension, which implement a set of functions
- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- Python has a ton of standard modules available.
- Standard modules can be imported the same way as we import our user-defined modules.

The *import* Statement

- Syntax:

  import module1[, module2[,... moduleN]

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.
- A module is loaded only once, regardless of the number of times it is imported.

Locating Modules

- When you import a module, the Python interpreter searches for the module in the following sequences –
  - The current directory.
  - If the module isn't 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/python/.

FROM…IMPORT * STATEMENT

- We can import specific names form a module without importing the module as a whole.
- Syntax :

```
from modname import *
```

- Example :

```
from example import add
```

Difference between import and from in Python

- Python's "import" loads a Python module into its own namespace, so that you have to add the module name followed by a dot in front of references to any names from the imported module that you refer to:

```
import math
print math.floor(x)
```

- "from" loads a Python module into the current namespace, so that you can refer to it without the need to mention the module name again:

```
from math import floor
print floor(x)
```

Files I/O

File Mode

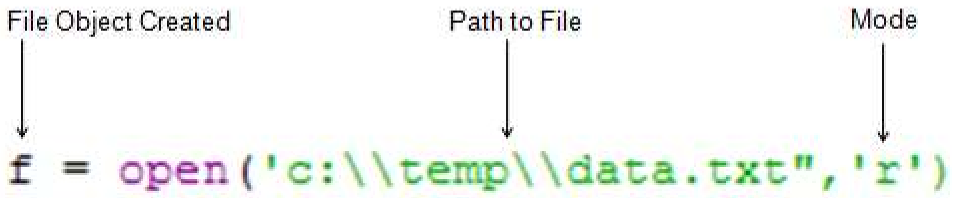
- **Read** : Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
- **Write** : Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
- **Append** : Opens a file for 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.

OPENING FILES

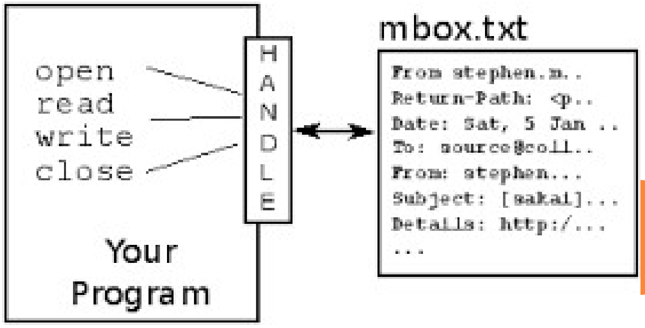
- Before you can read or write a file, you have to open it using Python's built-in*open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

- Syntax

File_object = open(file_name [, access_mode][, buffering])



```
f = open('c:\\temp\\data.txt",'r')
```



```
From stephen.m..
Return-Path: <p..
Date: Sat, 5 Jan ..
To: source@coll..
From: stephen...
Subject: [sakai]...
Details: http:/...
...
```

READ

Modes	Description
r	**Opens a file for reading only**. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
r+	**Opens a file for both reading and writing**. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.

Write

Modes	Description
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.
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.
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.
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.

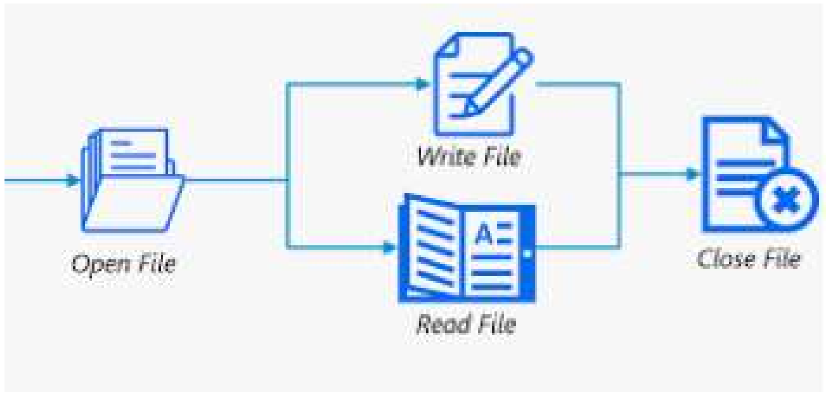
Append

Modes	Description
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.
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.
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.
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.

THE *CLOSE()* METHOD

- The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
- Syntax

```
fileObject.close()
```



The *file* Object Attributes

- Once a file is opened and you have one *file* object, you can get various information related to that file.

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

THE *READ()* METHOD

- 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.

THE *WRITE()* METHOD

- The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string –
- Syntax

```
fileObject.write(string);
```

File Positions

- The ***tell()*** method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.
- The ***seek(offset[, from])*** method changes the current file position. The *offset*argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.
- If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Renaming and 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.
- rename()

  ```
  os.rename(file_name, new_file_name)
  ```
- *remove()*

  ```
  os.remove(file_name)
  ```

OOP

Overview of OOP

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

Class and Object definition

- Class :

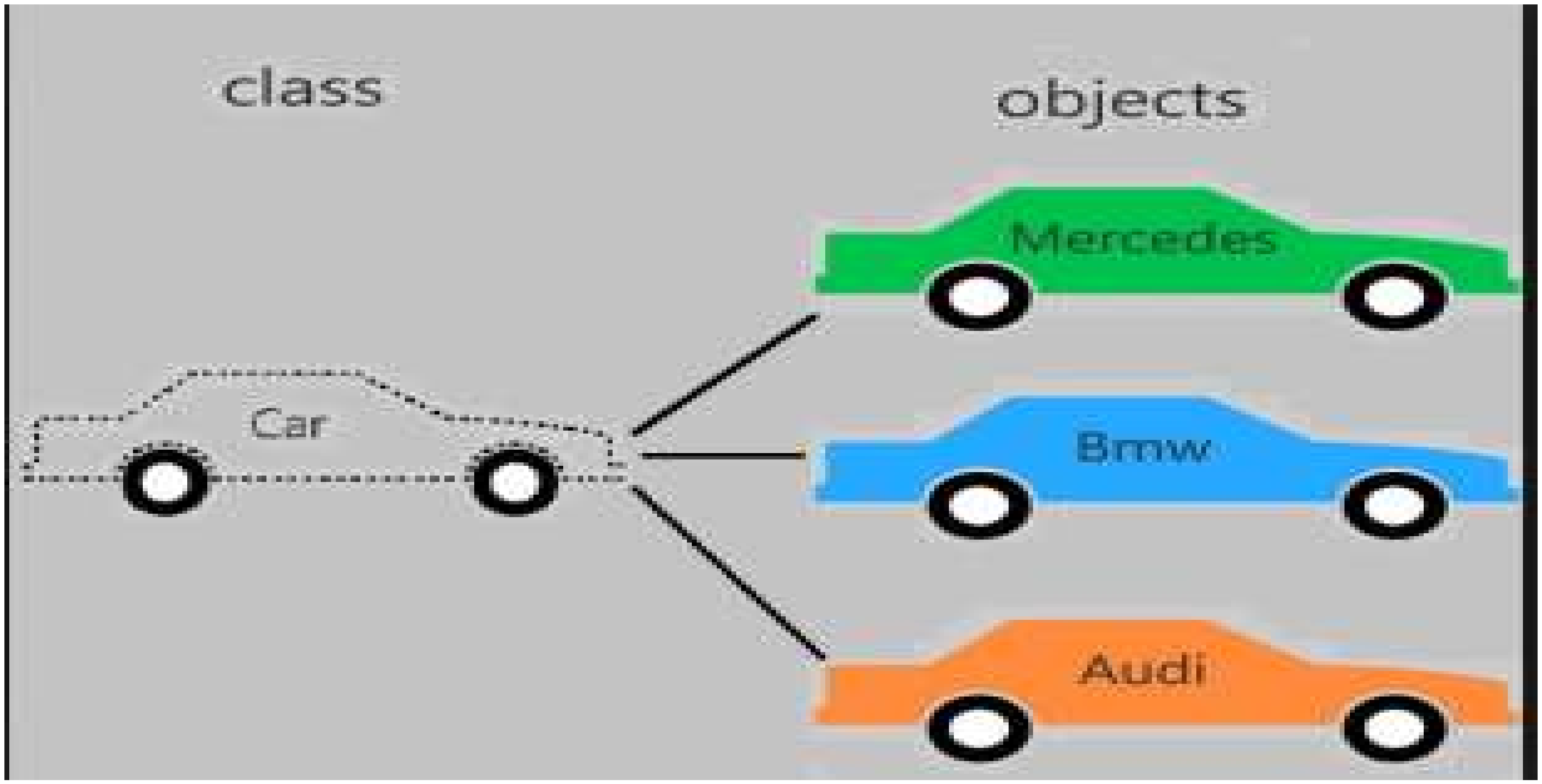
  Collection of objects. Logical entity. A class is blueprint for any functional entity which defines its properties and its functions.

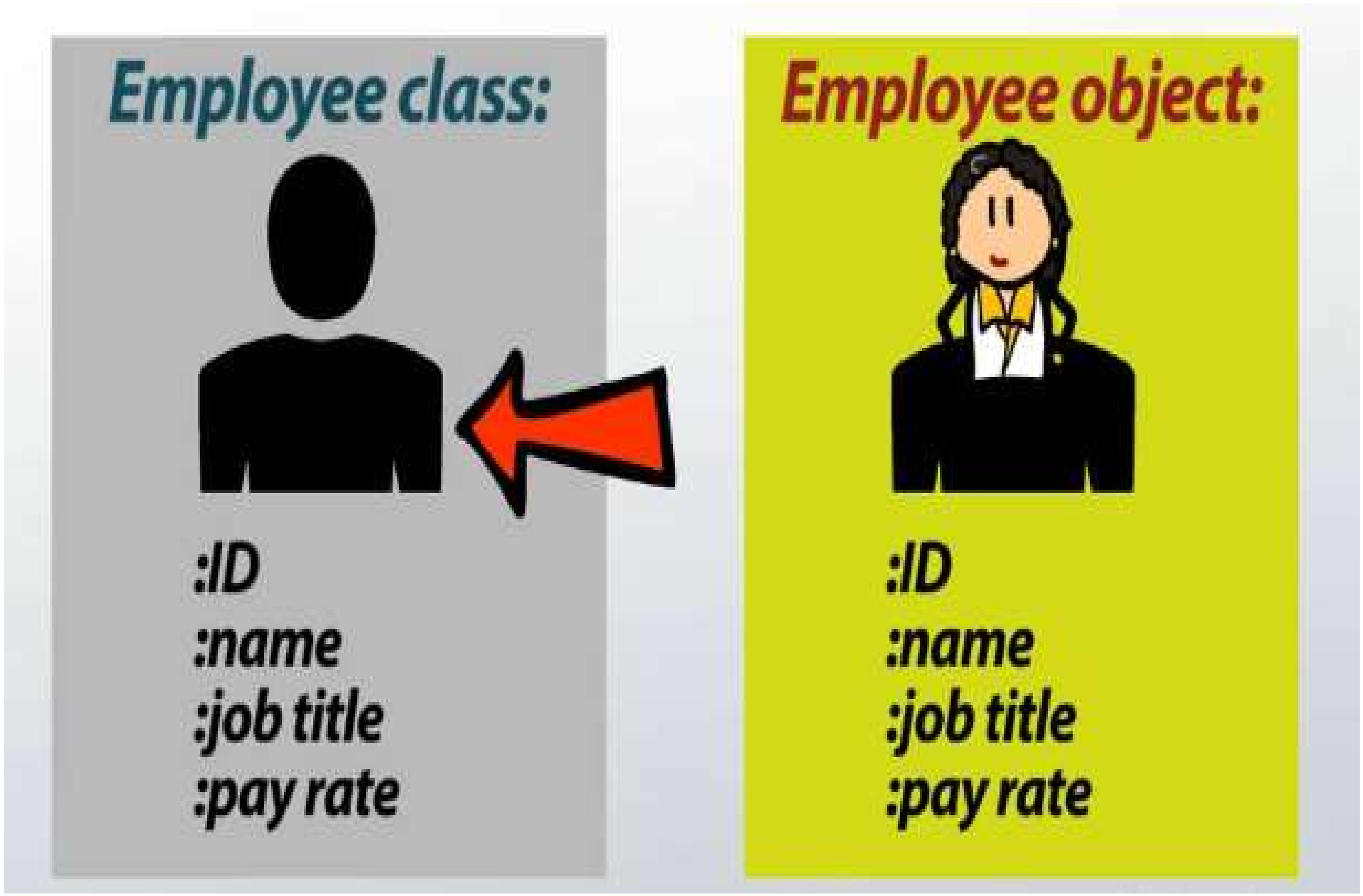
- Object :

  Logical and physical entity object is a real world entity. Object is an instance of class.

Class and Object

- Class is collection of similar objects




Employee class:
:ID
:name
:job title
:pay rate
Employee object:
:ID
:name
:job title
:pay rate

- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.

Creating Classes

- The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon.

- Syntax :

```
class ClassName:
   ' Optional class documentation string'
    class_suite
```

  - The class has a documentation string, which can be accessed via*ClassName.__doc__*.
  - The *class_suite* consists of all the component statements defining class variables and functions.

Class Example

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

  def displayCount() self:
   print "Total Employee %d" % Employee.empCount

  def displayEmployee(self):
    print "Name : ", self.name,  ", Salary: ", self.salary
```

Class Example

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as*Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.
- class can implement the special method__*del*__*()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Built-In Class Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –
- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.

Class Inheritance

- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.
- Syntax:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

DATA HIDING

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a **double underscore** prefix, and those attributes then are not be directly visible to outsiders.

Syntax :

```
    __var=value
```

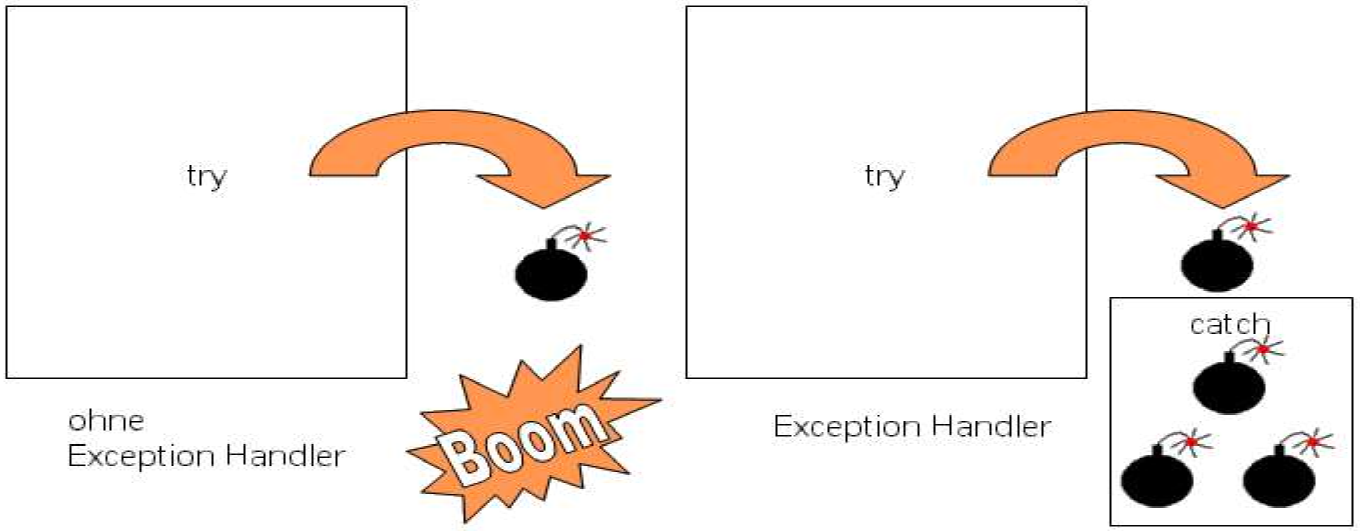
Exceptions Handling

What is Exception?

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Exception Handling in Python

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.



Exception Handling in Python

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Syntax

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
ArithmeticError	Base class for all errors that occur for numeric calculation.
FloatingPointError	Raised when a floating point calculation fails.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
NameError	Raised when an identifier is not found in the local or global namespace.

EXCEPTION NAME	DESCRIPTION
IOError IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Raised for operating system-related errors.
SyntaxError IndentationError	Raised when there is an error in Python syntax. Raised when indentation is not specified properly.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

The *except* Clause with No Exceptions

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

THE *EXCEPT* CLAUSE WITH MULTIPLE EXCEPTIONS

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception
list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

- You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

- Syntax :

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Argument of an Exception

- 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
- Syntax :

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

Raising an Exceptions

- You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.
- Syntax

```
raise [Exception [, args [, traceback]]]
```

- Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

Regular express

REGULAR EXPRESSIONS

Complex Regular Expression Symbols:

- **.** Matches any character
- ***** Matches one or more occurrences of that preceeding character
- **^** Anchors search to beginning of character string (line)
- **$** Anchors search to end of character string (line)
- **[..]** Matches any single character that belongs in square brackets. For opposite, use the notation **[^..]**
- **+** one or more character

Examples

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the beginning of the string
abc$	abc at the end of the string
a|b	either of a and b
^abc|abc$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does not contain any of a, b and c (such as defg)

Performing Queries with Regex in Python

The ‘re’ package provides several methods to actually perform queries on an input string. The methods that we will be discussing are

- re.match()
- re.search()
- re.findall()

Each of the methods accepts a regular expression, and string to scan for matches.

Find Using re.match – Matches Beginning

- Lets first take a look at the match() method. The way the match() method works is that it will only find matches if they occur **at the start of the string** being searched.
- So for example, calling match() on the string ‘dog cat dog’, looking for the pattern ‘dog’ will match:

```
>>> match = re.match(r'dog', 'dog cat dog')
>>> match.group()
 'dog‘
```

- But, if we call match() on the same string, looking for the pattern ‘cat’, we won’t:

```
>>> re.match(r'cat', 'dog cat dog')
>>>
```

re.search()

- The search() method is similar to match(), but search() doesn’t restrict us to only finding matches at the beginning of the string, so searching for ‘cat’ in our example string finds a match:

```
>>>x =re.search(r'cat', 'dog cat dog')
>>> match.group(0)
'cat‘
```

- The search() method, however, stops looking after it finds a match, so search()-ing for ‘dog’ in our example string only finds the first occurrence:

```
>>> x = re.search(r'dog', 'dog cat dog')
 >>> x.group(0)
 'dog'
```

FINDALL()

- The querying method that I use by far the most in python though is the findall() method. Rather than being returned match objects (we'll talk more about match objects in a little bit), when we call findall(), we **simply get a list of all matching patterns**. For me, this is just simpler. Calling findall() on our example string we get:

```
>>> re.findall(r'dog', 'dog cat dog')
['dog', 'dog']
>>> re.findall(r'cat', 'dog cat dog')
['cat']
```

GROUP BY NUMBER USING MATCH.GROUP

Grouping is the ability to address certain sub-parts of the entire regex match. We can define a group as a piece of the regular expression search string, and then individually address the corresponding content that was matched by this piece.

Let’s look at an example to see how this works:

```
>>> contactInfo = 'Doe, John: 555-1212‘
```

The string We just created resembles a snippet taken out of someones address book. We can match the line with a regular expression like this one:

```
>>> x=re.search(r'\w+, \w+: \S+', contactInfo)
```

Group by Number Using match.group

- By surrounding certain parts of the regular expression in parentheses (the '(' and ')' characters), we can group the content and then work with these individual groups.

```
 >>> x = re.search(r'(\w+), (\w+): (\S+)', contactInfo)
```

- These groups can be fetched using the match object's group() method. The groups are addressable numerically in the order that they appear, from left to right, in the regular expression (starting with group 1):

```
>>> x.group(1)
'Doe'
>>> x.group(2)
'John'
 >>> x.group(3)
'555-1212'
```

- The reason that the group numbering starts with group 1 is because group 0 is reserved to hold the entire match

```
>>> match.group(0)
 'Doe, John: 555-1212'
```

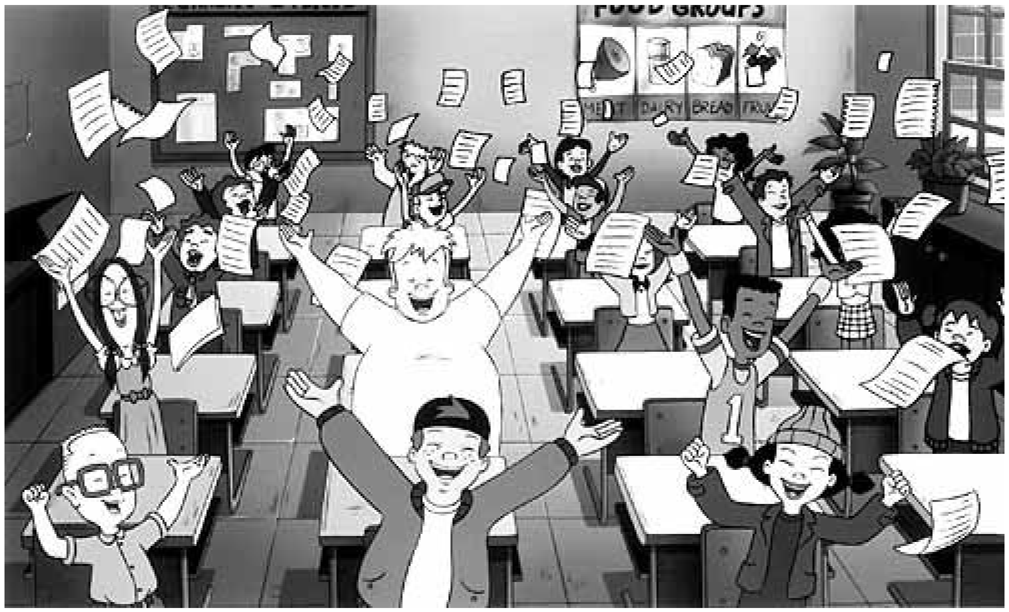
Grouping by Name Using match.group

- Sometimes, especially when a regular expression has a lot of groups, it is impractical to address each group by its number. Python also allows you to assign a name to a group using the following syntax:

```
>>> x = re.search(r'(?P<last>\w+), (?P<first>\w+): (?P<phone>\S+)', contactInfo)
```

- When can still fetch the grouped content using the group() method, but this time specifying the names we assigned the groups instead of the numbering we used before:

```
>>> x.group('last')
'Doe'
>>> x.group('first')
'John'
>>> x.group('phone')
'555-1212'
```

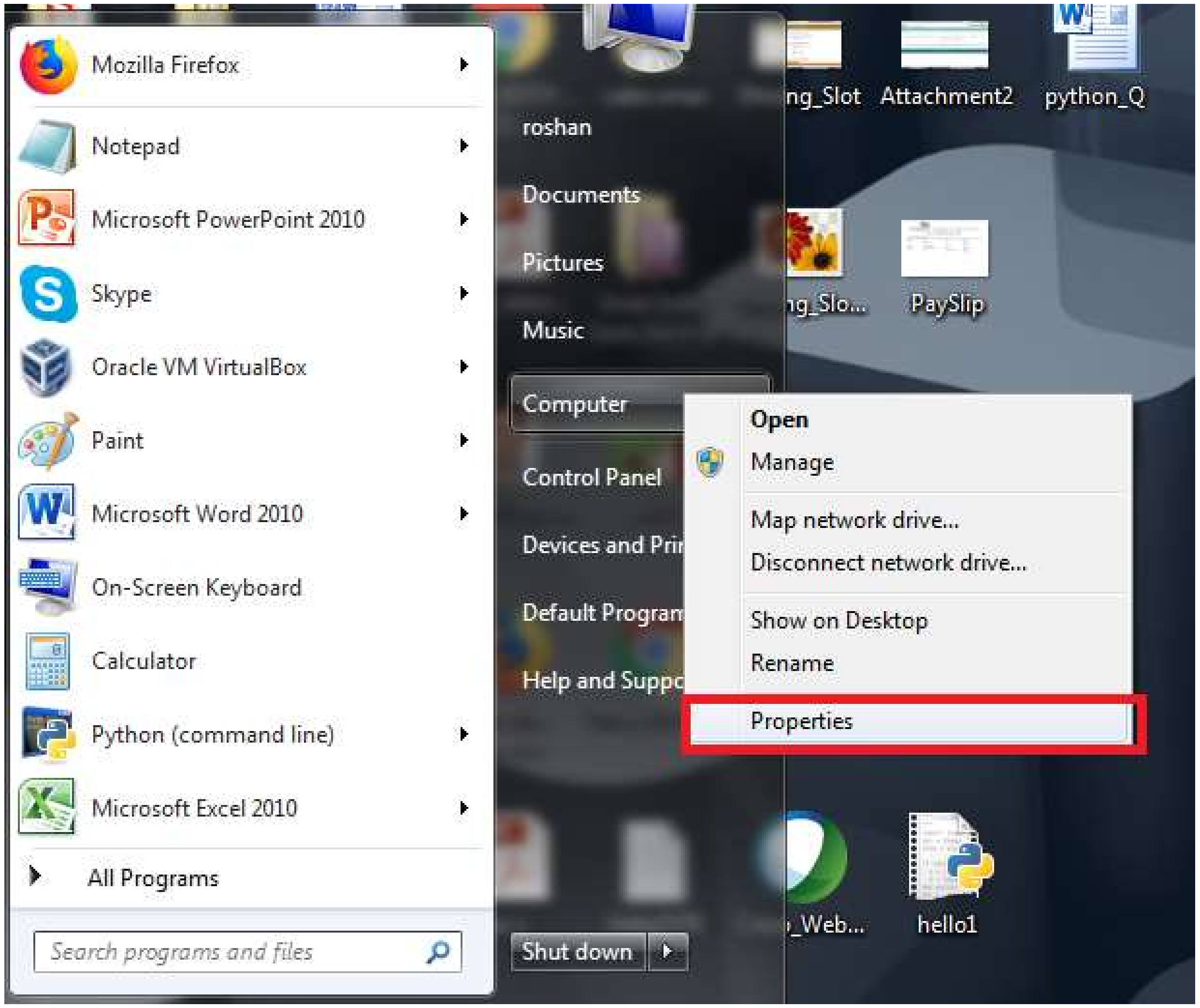


Hey …
I Am Python programmer……

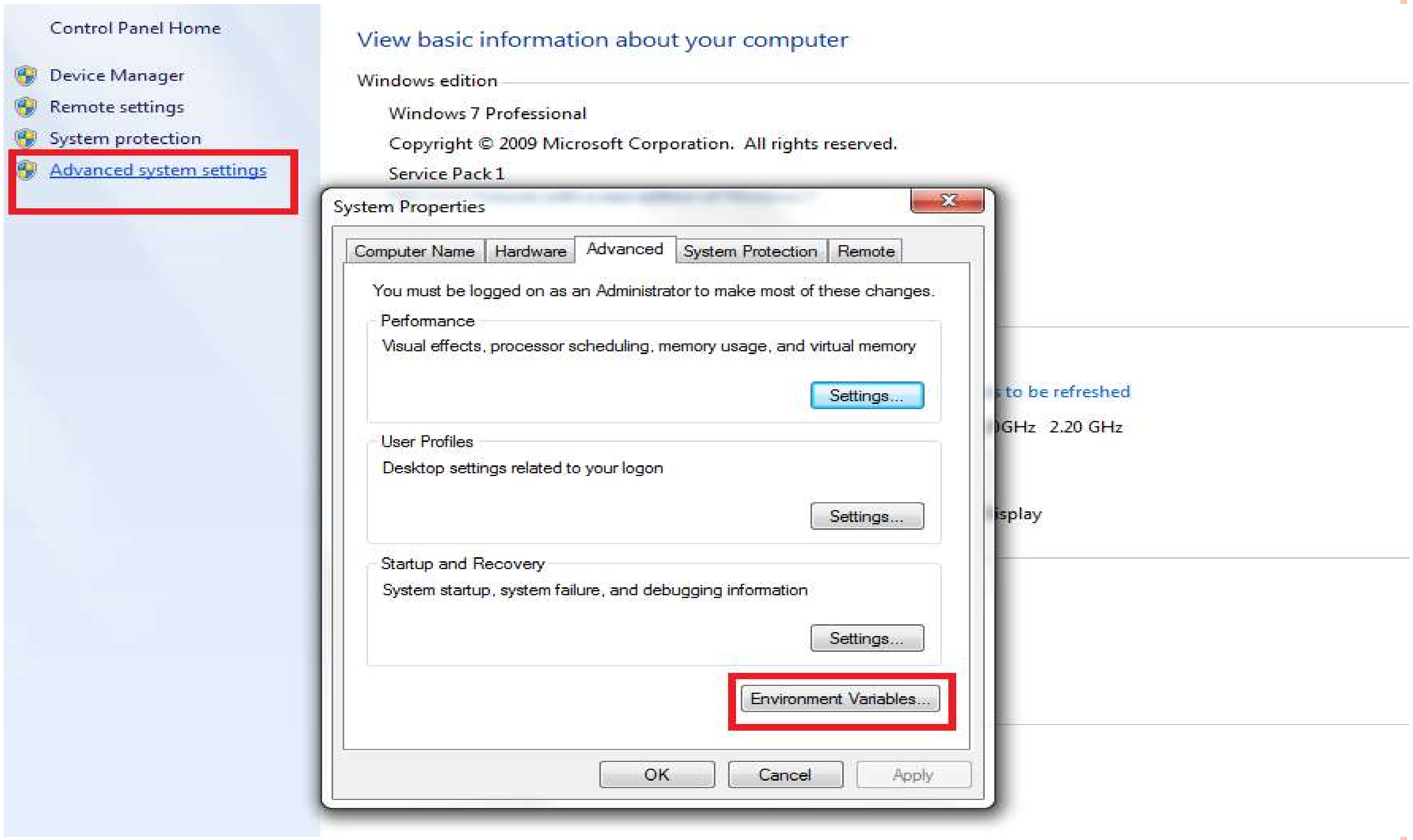
- Regular express

http://www.thegeekstuff.com/2014/07/python-regex-examples/

Python Setting



Python Settings



Control Panel Home

Device Manager

Remote settings

System protection

Advanced system settings

View basic information about your computer

Windows edition

Windows 7 Professional

Copyright © 2009 Microsoft Corporation. All rights reserved.

Service Pack 1

System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your logon

Settings...

Startup and Recovery

System startup, system failure, and debugging information

Settings...

Environment Variables...

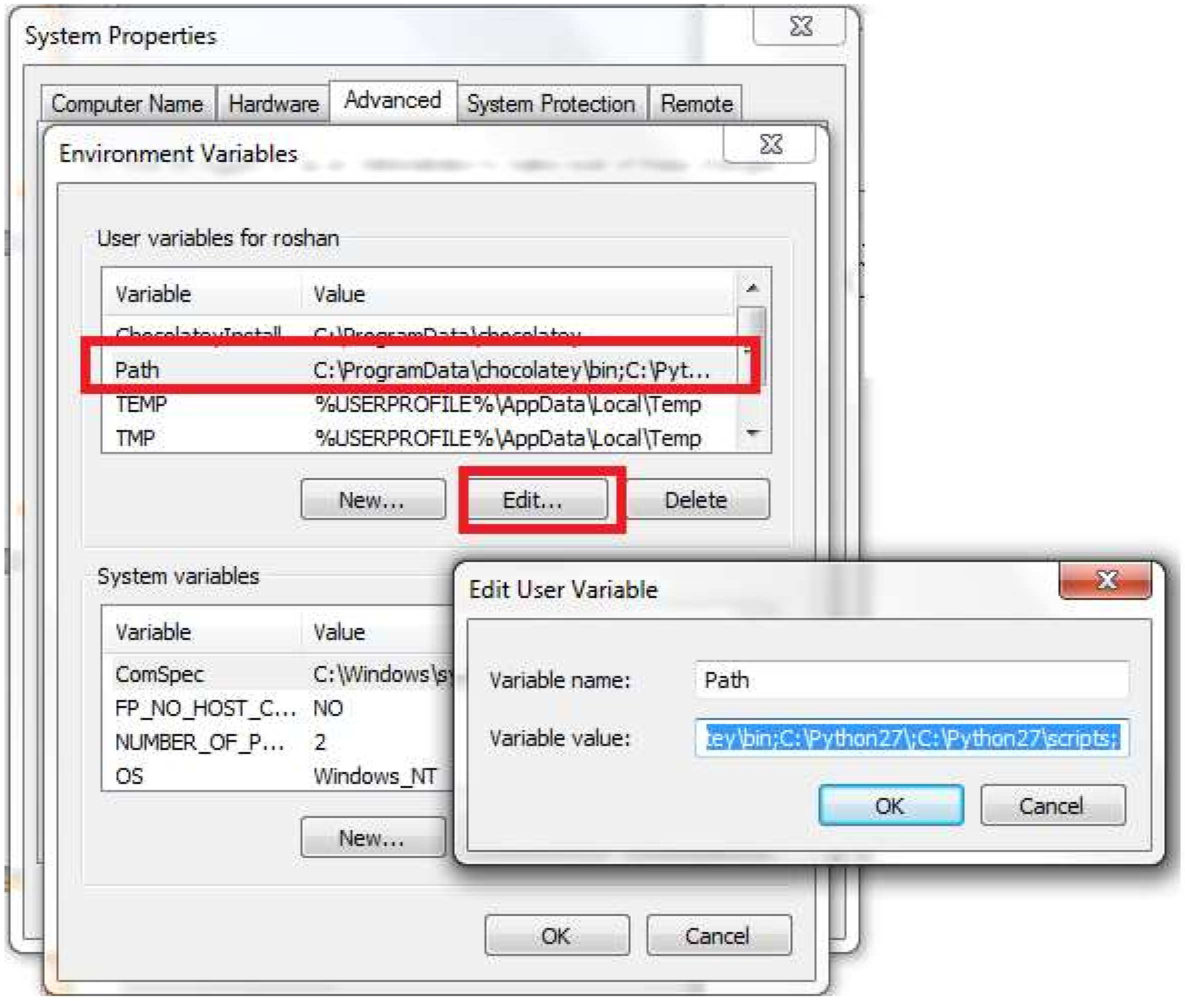
OK | Cancel | Apply

s to be refreshed

0GHz 2.20 GHz

isplay

Python Settings



System Properties

Environment Variables

User variables for roshan

Variable	Value
ChocolateyInstall	C:\ProgramData\chocolatey
Path	C:\ProgramData\chocolatey\bin;C:\Pyt...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\sy
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT

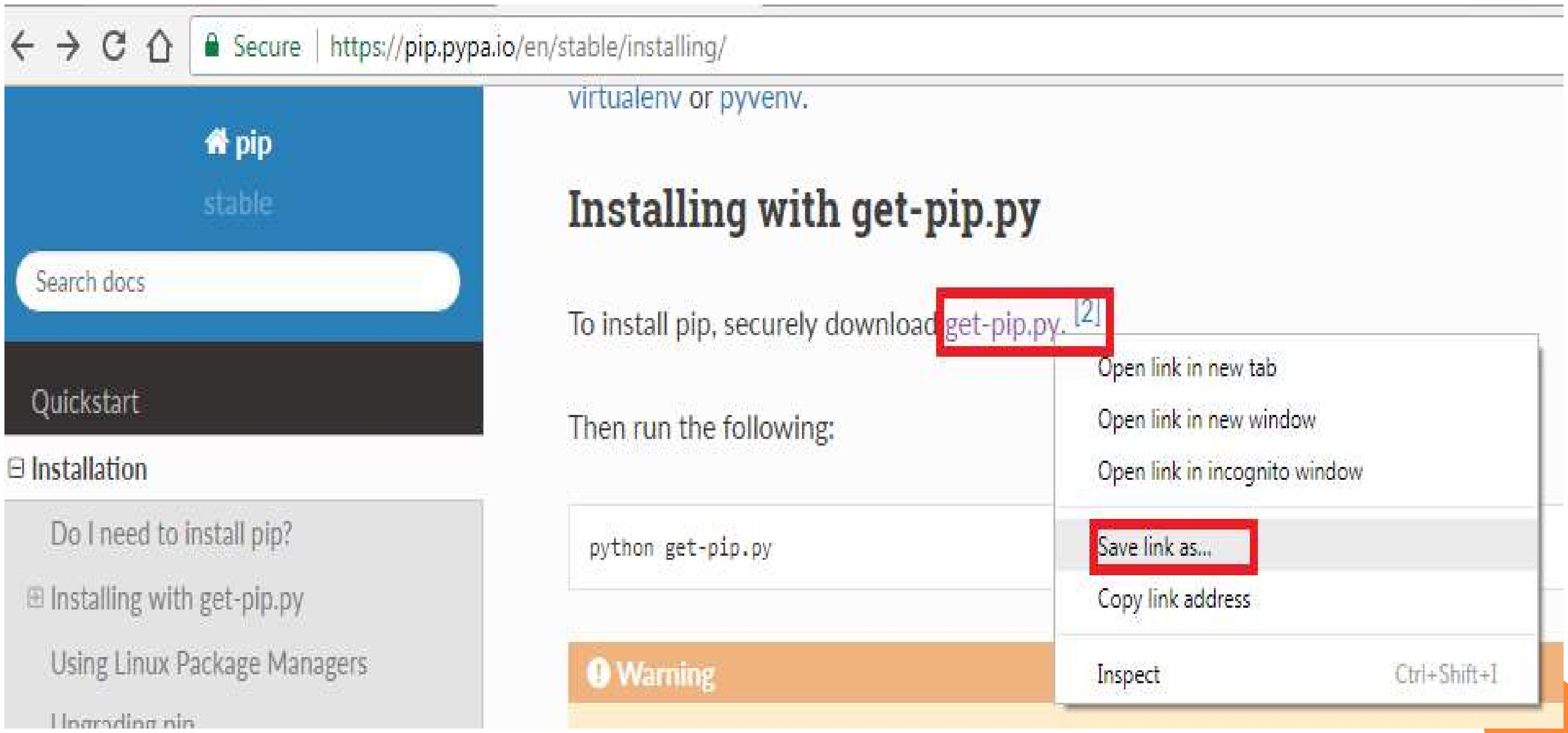
Edit User Variable

Variable name: Path

Variable value: tey\bin;C:\Python27\;C:\Python27\scripts;

Pip Installation on Windows

- https://pip.pypa.io/en/stable/installing/



Secure | https://pip.pypa.io/en/stable/installing/

virtualenv or pyvenv.

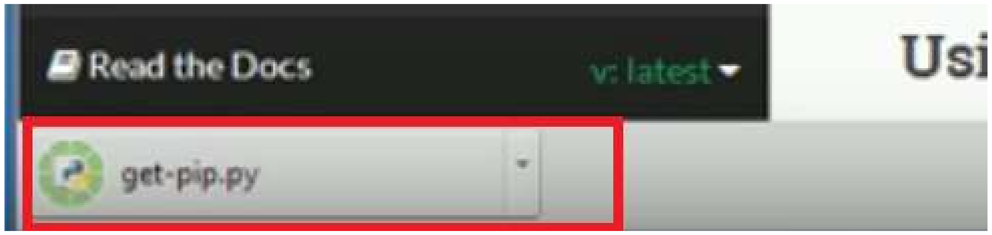
Installing with get-pip.py

To install pip, securely download get-pip.py. [2]

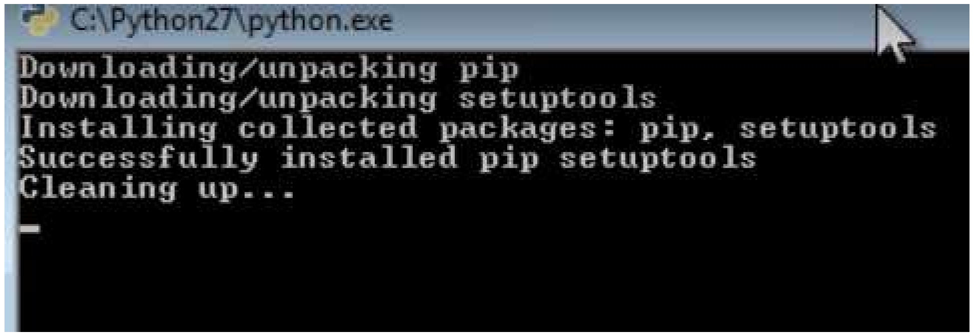
Then run the following:

```
python get-pip.py
```

- Click on downloaded file



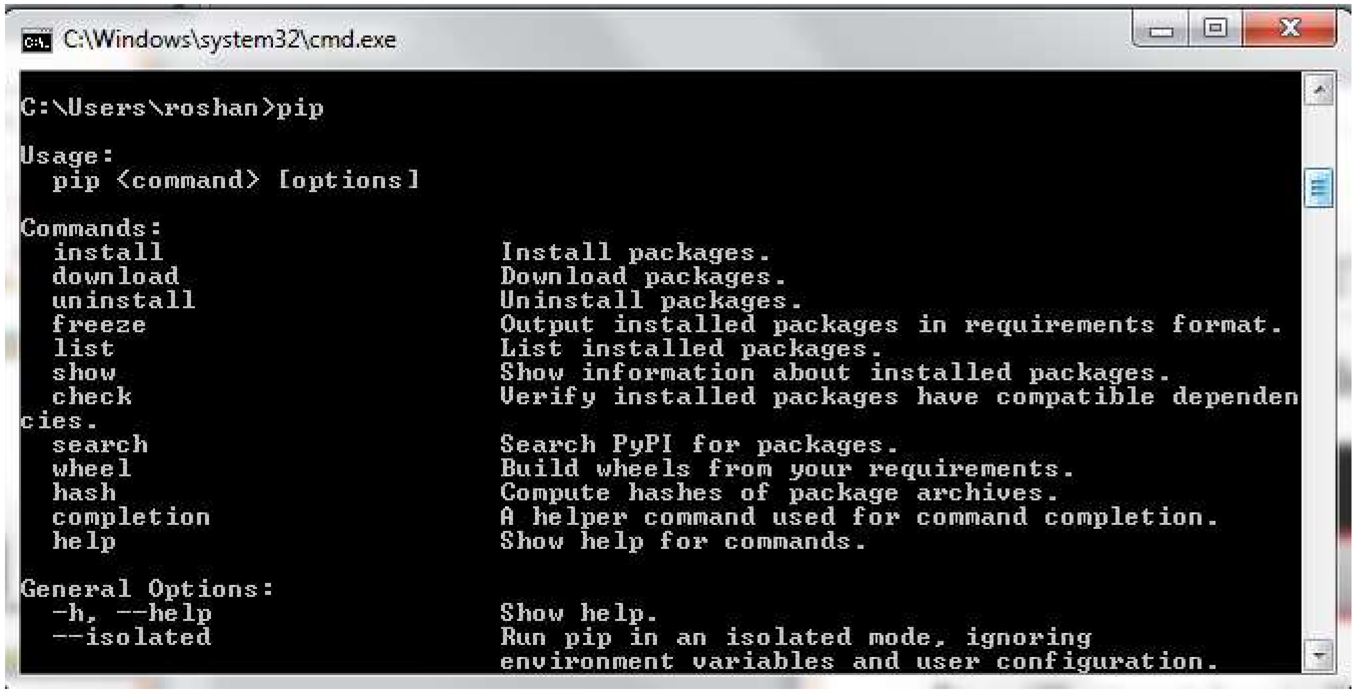
- It will start installation



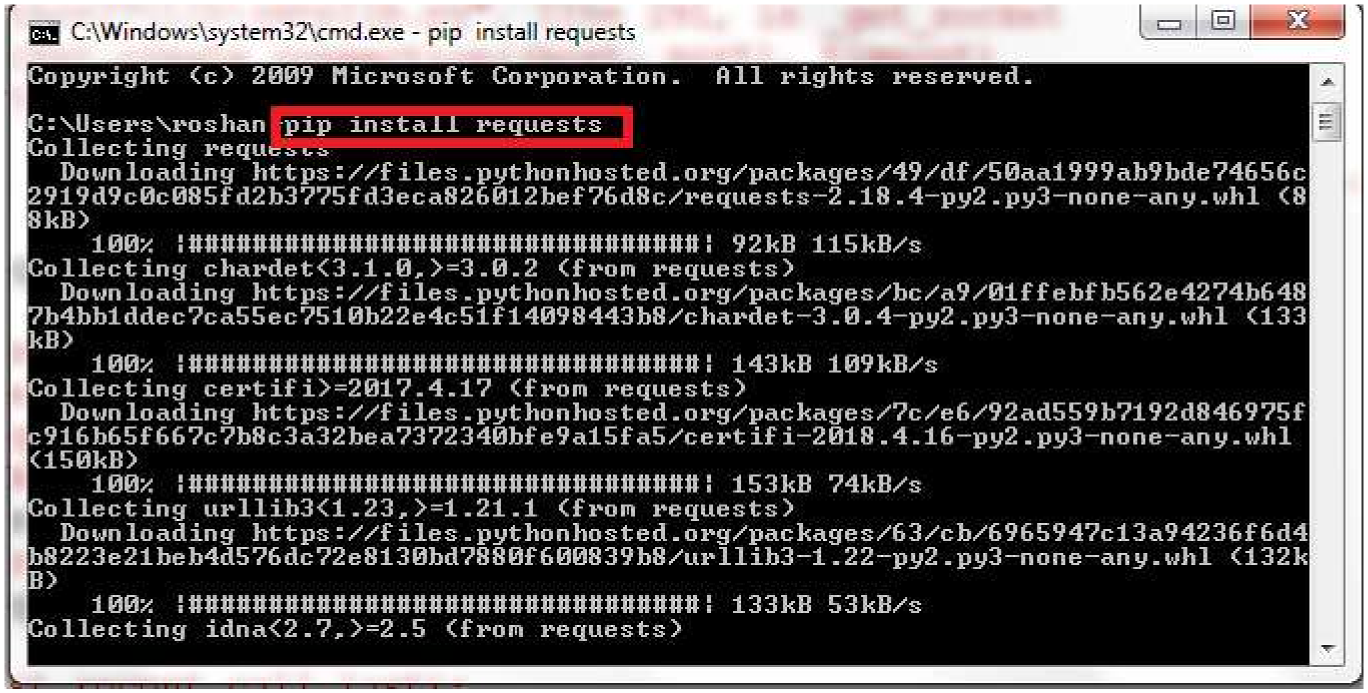
```
Downloading/unpacking pip
Downloading/unpacking setuptools
Installing collected packages: pip, setuptools
Successfully installed pip setuptools
Cleaning up...
_
```

- Add c:/python27/scripts in Environment variable

- Open command line and type “pip” you should be able to see pip help page



```
C:\Users\roshan>pip

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependen
cies.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  help                        Show help for commands.

General Options:
  -h, --help                  Show help.
  --isolated                  Run pip in an isolated mode, ignoring
                              environment variables and user configuration.
```



```
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\roshan pip install requests
Collecting requests
  Downloading https://files.pythonhosted.org/packages/49/df/50aa1999ab9bde74656c
2919d9c0c085fd2b3775fd3eca826012bef76d8c/requests-2.18.4-py2.py3-none-any.whl (8
8kB)
    100% |################################| 92kB 115kB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Downloading https://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b648
7b4bb1ddec7ca55ec7510b22e4c51f14098443b8/chardet-3.0.4-py2.py3-none-any.whl (133
kB)
    100% |################################| 143kB 109kB/s
Collecting certifi>=2017.4.17 (from requests)
  Downloading https://files.pythonhosted.org/packages/7c/e6/92ad559b7192d846975f
c916b65f667c7b8c3a32bea7372340bfe9a15fa5/certifi-2018.4.16-py2.py3-none-any.whl
(150kB)
    100% |################################| 153kB 74kB/s
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Downloading https://files.pythonhosted.org/packages/63/cb/6965947c13a94236f6d4
b8223e21beb4d576dc72e8130bd7880f600839b8/urllib3-1.22-py2.py3-none-any.whl (132k
B)
    100% |################################| 133kB 53kB/s
Collecting idna<2.7,>=2.5 (from requests)
```

Anaconda installation

- https://docs.anaconda.com/anaconda/install/windows/