What is Python…?

- Python is a general purpose programming language that is often applied in scripting roles.
- So, Python is programming language as well as scripting language.
- Python is also called as Interpreted language

Differences between program and scripting language

Program

- a **program is executed** *(i.e. the source is first compiled, and the result of that compilation is expected)*
- A "program" in general, is **a sequence of instructions written so that a computer can perform certain task**.

Scripting

- a **script is interpreted**
- A "script" is code written in a scripting language. A scripting language is nothing but **a type of programming language in which we can write code to control another** software application.

History

- Invented in the Netherlands, early 90s by Guido van Rossum
- Python was conceived in the late 1980s and its implementation was started in December 1989
- Guido Van Rossum is fan of 'Monty Python's Flying Circus', this is a famous TV show in Netherlands
- Named after Monty Python
- Open sourced from the beginning

Scope of Python

- Science
  - Bioinformatics
- System Administration
  - -Unix
  - -Web logic
  - -Web sphere
- Web Application Development
  - -CGI
  - -Jython – Servlets
- Testing scripts

Features of Python:

Why do people use Python…?

The following primary factors cited by Python users seem to be these:

- Python is object-oriented

  Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance.

- Indentation

  Indentation is one of the greatest future in Python.

- It's free (open source)

  Downloading and installing Python is free and easy

  Source code is easily accessible

- It's powerful
  - Dynamic typing
  - Built-in types and tools
  - Library utilities
  - Third party utilities (e.g. Numeric, NumPy, SciPy)
  - Automatic memory management
- It's portable
  - Python runs virtually every major platform used today
  - As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform.

- It's mixable
  - Python can be linked to components written in other languages easily
  - Linking to fast, compiled code is useful to computationally intensive problems
  - - Python/C integration is quite common
- It's easy to use
  - No intermediate compile and link steps as in C/ C++
  - Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
  - This gives Python the development speed of an interpreter without the performance loss inherent in purely interpreted languages
- It's easy to learn
  - Structure and syntax are pretty intuitive and easy to grasp

Installing Python

- Python is pre-installed on most Unix systems, including Linux and MAC OS X

- But for in Windows Operating Systems , user can download from the https://www.python.org/downloads/
  - from the above link download latest version of python IDE and install, recent version is 3.4.1 but most of them uses version 2.7.7 only

What can I do with Python…?

- System programming
- Graphical User Interface Programming
- Internet Scripting
- Component Integration
- Database Programming
- Gaming, Images, XML , Robot and more

A Sample Code

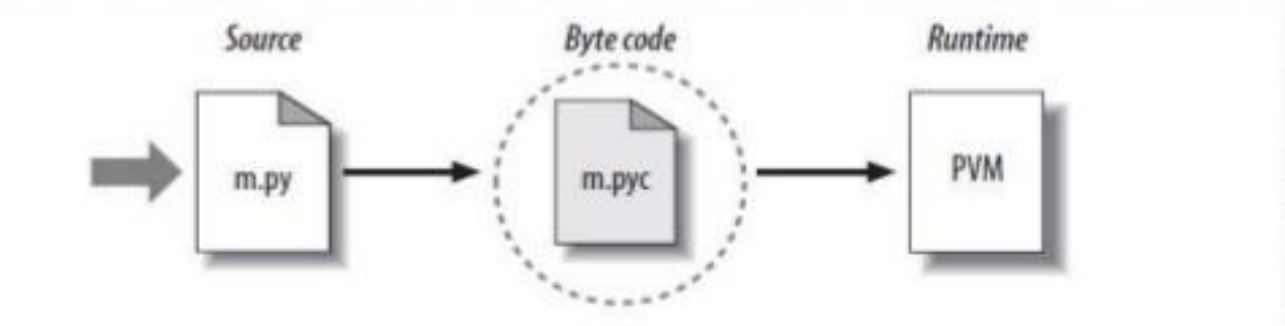
```
x = 34 - 23          # A comment.
y = "Hello"          # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"  # String concat.
print x
print y
```

Enough to understand the code

- Indentation matters to code meaning
  - Block structure indicated by indentation
- First assignment to a variable creates it
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- Assignment is = and comparison is ==
- For numbers + - * / % are as expected
  - Special use of + for string concatenation and % for string formatting (as in C's printf)
- Logical operators are words (and, or, not) *not* symbols
- The basic printing command is print

Python Code Execution

- Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.



Source code extension is **.py**
Byte code extension is **.pyc** (compiled python code)

Running Python

Once you're inside the Python interpreter, type in commands at will.

- Examples:

```
>>> print 'Hello world'
Hello world
# Relevant output is displayed on subsequent lines without the >>> symbol
>>> x = [0,1,2]
# Quantities stored in memory are not displayed by default
>>> x
# If a quantity is stored in memory, typing its name will display it
[0,1,2]
>>> 2+3
5
```

Python Statement

- Instructions that a Python interpreter can execute are called ***statements***.
- For example, a = 1 is an assignment statement. if statement, for statement, while statement etc. are other kinds of statements.

Multi-line statement

- In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the ***line continuation character (***\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

- This is explicit line continuation.
- In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }.
- For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
     4 + 5 + 6 +
     7 + 8 + 9)
```

- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
          'green']
```

- We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

Python Indentation

- Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
- A code block (body of a function,loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

- Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

- Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing. But the former style is clearer.

- Incorrect indentation will result into IndentationError.

Python Comments

- Comments are very important while writing a program. ***It describes what's going on inside a program*** so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.
- ***In Python, we use the hash (#) symbol to start writing a comment***.

- ***It extends up to the newline character***. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#This is a comment
#print out Hello
print('Hello')
```

Python Keywords :

- Keywords are the **reserved words** in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7. This number can vary slightly in the course of time.
- All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

False class finally is return

none continue for lambda try

True def from nonlocal while

and del global not with

as elif if or yield

assert else import pass

break except in raise

Python Identifiers :

- ***An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.***

Rules for writing identifiers

- Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore _. Names like myClass, var_1 and print_this_to_screen, all are valid example.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
- Keywords cannot be used as identifiers.

Example:

```
>>>global = 1
File"<interactive input>", line 1
global = 1
         ^
SyntaxError: invalid syntax
```

- We cannot use special symbols like **!**, **@**, **#**, **$**, **%** etc. in our identifier.

```
>>> a@ = 0
File"<interactive input>", line 1
   a@ = 0
    ^
SyntaxError: invalid syntax
```

- Identifier can be of any length.

Python's Core Data types:

Object type------ Example literals/creation

- Numbers---------- 1234, 3.1415, 3+4j, Decimal, Fraction
- Strings ---------- 'spam', "guido's", b'a\x01c'
- Lists ----------[1, [2, 'three'], 4]
- Dictionaries------- {'food': 'spam', 'taste': 'yum'}
- Tuples ----------(1, 'spam', 4, 'U')
- Files---------- myfile = open('eggs', 'r')
- Sets---------- set('abc'), {'a', 'b', 'c'}
- Other core types--- Booleans, types, None
- Program unit types--- Functions, modules, classes
- Implementation-related types----- Compiled code, stack tracebacks

Numbers :

- Numbers are fairly straightforward.
- Python's core objects set includes the usual suspects: ***integers*** (numbers without a fractional part), ***floating-point numbers*** (roughly, numbers with a decimal point in them), and more ***exotic numeric types*** (complex numbers with imaginary parts, fixed-precision decimals, rational fractions with numerator and denominator, and full-featured sets).
- Python's basic number types are, well basic.

- Numbers in Python support the normal mathematical operations. For instance, the plus sign **(+)** performs addition, a star **(*)** is used for multiplication, and two stars **(**)** are used for exponentiation.

```
>>> 123 + 222    # Integer addition
345
>>> 1.5 * 4      # Floating-point multiplication
6.0
>>> 2 ** 100     # 2 to the power 100
1267650600228229401496703205376
```

- ***Python 3.0's integer type automatically provides extra precision for large numbers like this when needed***

Examples:

```
>>> 3.1415 * 2   # repr: as code
6.2830000000000004
>>> print(3.1415 * 2)    # str: user-friendly
6.283
```

- Two ways to print every object: with full precision (as in the first result shown here), and in a userfriendly form (as in the second).

- Besides expressions, there are a handful of useful ***numeric modules*** that ship with Python—*modules are just packages of additional tools that we import to use:*

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

Strings :

- To create string literals, enclose them in single, double, or triple quotes as follows:

```
a = "Hello World"
b = 'Python is groovy'
c = """Computer says 'No'"""
```

- The ***same type of quote used to start a string must be used to terminate it***. Triple quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line

- Triple-quoted strings are useful when the contents of a string literal span multiplelines of text such as the following:

```
print '''Content-type: text/html
<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
'''
```

- Strings are ***stored as sequences of characters*** indexed by ***integers, starting at zero***. To extract a single character, use the indexing operator s[i] like this:

```
a = "Hello World"
b = a[4]     # b = 'o'
```

- To ***extract a substring***, use the ***slicing operator*** **s[i:j]**.This extracts all characters from s whose index k is in the range **i <= k < j**. If either index is omitted, the beginning or end of the string is assumed, respectively:

```
c = a[:5]     # c = "Hello"
d = a[6:]     # d = "World"
e = a[3:8]    # e = "lo Wo"
```

- Strings are ***concatenated*** with the plus **(+)** operator:

```
g = a + " This is a test"
```

- Python ***never implicitly interprets the contents of a string as numerical data*** (i.e., as in other languages such as Perl or PHP). For example, + always concatenates strings:

```
x = "37"
y = "42"
z = x + y     # z = "3742" (String Concatenation)
```

- To perform mathematical calculations, strings first have to be converted into a numeric value using a function such as int() or float(). For example:

```
z = int(x) + int(y)        # z = 79 (Integer +)
```

- Non-string values can be converted into a string representation by using the str(),

repr(), or format() function. Here's an example:

```
s = "The value of x is " + str(x)
s = "The value of x is " + repr(x)
s = "The value of x is " + format(x,"4d")
```

- Although str() and repr() both create strings, their output is usually slightly different. str() ***produces the output that you get when you use the print statement***, whereas repr() ***creates a string that you type into a program to exactly represent the value of an object***.

For example:

>>> **x = 3.4**

>>> **str(x)**

'3.4'

>>> **repr(x)**

'3.3999999999999999'

>>>

- The ***format() function is used to convert a value to a string with a specific formatting applied***. For example:

>>> **format(x,"0.5f")**

'3.40000'

>>>

For example:

```
# Python3 program to demonstarte
# the str.format() method

# using format option in a simple string
   print ("{}, A computer science portal for students."
                     .format(“PYTHON"))

# using format option for a
# value stored in a variable
str = "This article is written in {}"
print (str.format("Python"))

# formatting a string using a numeric constant
print ("Hello, I am {} years old !".format(18))
```

Output :

```
PYTHON, A computer science portal for students.
This article is written in Python
 Hello, I am 18 years old!
```

Lists:

- ***Lists are sequences of arbitrary objects***. *You create a list by enclosing values in* ***square*** **brackets**, as follows:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

- Lists are ***indexed by integers***, starting with zero. ***Use the indexing operator to access and modify individual items of the list***:

```
a = names[2]          # Returns the third item of the list, "Ann"
names[0] = "Jeff"  # Changes the first item to "Jeff"
```

- To ***append new items*** to the end of a list, use the **append()** method and **extend()** method to add several items:

```
names.append("Paula")
```

- To ***insert an item*** into the middle of a list, use the **insert()** method:

```
names.insert(2, "Thomas")
```

- You can extract or reassign a portion of a list by using the slicing operator:

b = names[0:2] *# Returns* ["Jeff", "Mark"]

c = names[2:] *# Returns* ["Thomas", "Ann", "Phil", "Paula"]

names[1] = 'Jeff' *# Replace the 2nd item in names with* 'Jeff'

names[0:2] = ['Dave','Mark','Jeff'] *# Replace the first two items of the list with the list on the right.*

- Use the ***plus (+) operator*** to concatenate lists:

```
a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]
```

- An ***empty list is created*** in one of two ways:

```
names = []          # An empty list
names = list()      # An empty list
```

- ***Lists can contain any kind of Python object***, including other lists, as in the following

Example:

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

- Items contained in nested lists are accessed by applying more than one indexing operation, as follows:

```
a[1]          # Returns "Dave"
a[3][2]       # Returns 9
a[3][3][1]    # Returns 101
```

Different List Methods -

- **list(s)-**> Converts s to a list.
- **s.append(x) -**> Appends a new element, x, to the end of s.
- **s.extend(t) -**> Appends a new list, t, to the end of s.
- **s.count(x) -**> Counts occurrences of x in s.
- **s.index(x [,start [,stop]]) -**> Returns the smallest i where s[i]==x. start and stop optionally specify the starting and ending index for the search.
- **s.insert(i,x) -**> Inserts x at index i.
- **s.pop([i]) -**> Returns the element i and removes it from the list. If i is omitted, the last element is returned.
- **s.remove(x) -**> Searches for x and removes it from s.
- **s.reverse() -**> Reverses items of s in place.

- **s.sort([key [, reverse]])** -> Sorts items of s in place. **key** is a key function. **reverse** is a flag that sorts the list in reverse order. key and reverse should always be specified as keyword arguments.

Tuples:

- To ***create simple data structures, you can pack a collection of values together into a single object*** using a ***tuple***. *You create a tuple by enclosing a group of values in parentheses* like this:

stock = (‘XYZ', 100, 490.10)

address = ('www.python.org', 80)

person = (first_name, last_name, phone)

- Python often recognizes that a ***tuple is intended even if the parentheses are missing***:

stock = ‘XYZ', 100, 490.10

address = 'www.python.org',80

person = first_name, last_name, phone

Tuples: contd…

- For completeness, 0- and 1-element tuples can be defined, but have special syntax:

```
a = () # 0-tuple (empty tuple)
b = (item,) # 1-tuple (note the trailing comma)
c = item, # 1-tuple (note the trailing comma)
```

- The values in a tuple can be ***extracted by numerical index*** just like a list. However, it is more common to unpack tuples into a set of variables like this:

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

- Although tuples ***support most of the same operations as lists*** (such as indexing, slicing, and concatenation), the ***contents of a tuple cannot be modified after creation*** (that is, you cannot replace, delete, or append new elements to an existing tuple).

- This reflects the fact that a ***tuple is best viewed as a single object consisting of several parts***, not as a collection of distinct objects to which you might insert or remove items.

- Because there is so much overlap between tuples and lists, some ***programmers*** are inclined to ignore tuples altogether and ***simply use lists because they seem to be more flexible***.

- Although this works***, it wastes memory if your program is going to create a large number of small lists*** (that is, each containing fewer than a dozen items).This is because lists slightly over allocate memory to optimize the performance of operations that add new items. Because ***tuples are immutable, they use a more compact representation where there is no extra space***.

- For example, this program shows how you might read a file consisting of different columns of data separated by commas:

```
# File containing lines of the form "name,shares,price"
filename = "portfolio.csv"       # comma seperated values
portfolio = []
for line in open(filename):
fields = line.split(",") # Split each line into a list
name = fields[0] # Extract and convert individual fields
shares = int(fields[1])
price = float(fields[2])
stock = (name,shares,price) # Create a tuple (name, shares, price)
portfolio.append(stock) # Append to list of records
```

- The ***split() method of strings splits a string into a list of fields separated by the given delimiter character***. The resulting portfolio data structure created by this program looks like a ***two-dimension array of rows and columns***. ***Each row is represented by a tuple*** and can be accessed as follows:

>>> portfolio[0]

(‘XYZ', 100, 490.10)

>>> **portfolio[1]**

(‘PQR', 50, 54.23)

>>>

Individual items of data can be accessed like this:

>>> **portfolio[1][1]**

50

>>> **portfolio[1][2]**

54.23

Dictionaries:

- **A *dictionary is an associative array or hash table that contains objects indexed by keys.***
- You create a dictionary by enclosing the values in curly braces ({ }), like this:

```
stock = {
        "name" : “KLPT",
        "shares" : 100,
        "price" : 490.10
      }
```

- ***To access members of a dictionary, use the key-indexing operator as follows***:

```
name = stock["name"]
value = stock["shares"] * shares["price"]
```

- ***Inserting or modifying objects works like this***:

stock["shares"] = 75

stock["date"] = "June 7, 2007“

- A dictionary is a ***useful way to define*** an object that consists of named fields.
- However, dictionaries are also used as a ***container for performing fast lookups on unordered data***. For example, here’s a dictionary of stock prices:

```
prices = {
    “KLPT" : 490.10,
    "AAPL" : 123.50,
    "IBM" : 91.50,
    "MSFT" : 52.13
}
```

- An ***empty dictionary*** is created in one of two ways:

```
prices = {} # An empty dict
prices = dict() # An empty dict
```

- Dictionary membership is tested with the **in** operator, as in the following example:

```
if "SCOT" in prices:
    p = prices["SCOT"]
else:
    p = 0.0
```

- This particular sequence of steps can also be performed more compactly as follows:

```
p = prices.get("SCOT",0.0)
```

- To obtain a ***list of dictionary keys***, convert a dictionary to a list:

```
syms = list(prices)
# syms = ["AAPL", "MSFT", "IBM", “KLPT"]
```

- Use the **del** statement ***to remove an element*** of a dictionary:

```
del prices["MSFT"]
```

- Dictionaries are probably the ***most finely tuned data type*** in the Python interpreter.