Introduction to Python

What is Python?

Python is a multi-paradigm high-level general purpose scripting language.

Python supports multiple programming paradigms (models) like structured programming, object-oriented programming, and functional programming. So, it is called a multi-paradigm language.

Python is user friendly and can be used to develop application software like text editors, music players, etc. So, it is called a high-level language.

Python has several built-in libraries. There are also many third-party libraries. With the help of all these libraries we develop almost anything with Python. So, it is called a general purpose language.

Finally, Python programs are executed line by line (interpreted). So, it is called a scripting language.

Who Created Python?

Python was created by Guido Van Rossum. He worked at Google from 2005-2012. From 2013 onwards he is working at Dropbox.

Guido Van Rossum was fond of a British comedian group named Monty Python. Their BBC series was Flying Circus. Rossum named the language after them (Monty Python).

History of Python

- Python implementation began in 1989.
- Python is a successor to ABC language (itself inspired by SETL).
- First official release on 20 Feb 1991.
- Python 2.0 was released on 16 Oct 2000.
- Python 3.0 was released on 3 Dec 2008.
- Latest stable release is Python 3.6.1 released in Mar. 2017.

Need for Python Programming

Following are some of the main reasons for learning Python language:

Software quality: Python code is designed to be readable, and hence reusable and maintainable much more than traditional scripting languages. Python has deep support for more advanced software reuse mechanisms, such as object-oriented (OO) and function programming.

Developer productivity: Python code is typically one-third to one-fifth the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact.

Program portability: Most Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines.

Support Libraries: Python comes with a large collection of prebuilt and portable functionality, known as the standard library. This library supports an array of application-level programming tasks, from text pattern matching to network scripting.

Python's third-party domain offers tools for website construction, numeric programming, serial port access, game development, and much more. The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

Component Integration: Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components.

Applications of Python

Systems Programming: Python's built-in interfaces to operating-system services make it ideal for writing portable, maintainable system-administration tools and utilities (sometimes called shell tools). Python programs can search files and directory trees, launch other programs, do parallel processing with processes and threads, and so on.

GUIs: Python's simplicity and rapid turnaround also make it a good match for graphical user interface programming on the desktop. Python comes with a standard object-oriented interface to the Tk GUI API called tkinter (Tkinter in 2.X) that allows Python programs to implement portable GUIs with a native look and feel.

Internet Scripting: Python comes with standard Internet modules that allow Python programs to perform a wide variety of networking tasks, in client and server modes. Scripts can communicate over sockets; extract form information sent to server-side CGI scripts; transfer files by FTP; parse and generate XML and JSON documents; send, receive, compose, and parse email; fetch web pages by URLs; parse the HTML of fetched web pages; communicate over XML-RPC, SOAP, and Telnet; and more. Python's libraries make these tasks remarkably simple.

Component Integration: Python's ability to be extended by and embedded in C and C++ systems makes it useful as a flexible glue language for scripting the behavior of other systems and components.

Tools such as the SWIG and SIP code generators can automate much of the work needed to link compiled components into Python for use in scripts, and the Cython system allows coders to mix Python and C-like code.

Database Programming: For traditional database demands, there are Python interfaces to all commonly used relational database systems like Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite, and more. The Python world has also defined a portable database API for accessing SQL database systems from Python scripts, which looks the same on a variety of underlying database systems.

In the non-SQL department, Python's standard pickle module provides a simple object persistence system—it allows programs to easily save and restore entire Python objects to files and file-like objects.

Rapid Prototyping: To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially, and then move selected components to a compiled language such as C or C++ for delivery.

Numeric and Scientific Computing: Python is also heavily used in numeric programming. Prominent here, the NumPy high-performance numeric programming extension for Python includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more.

Features of Python

Object Oriented and Functional: Python is an object-oriented language, from the ground up. Its class model supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet, in the context of Python's simple syntax and typing, OOP is remarkably easy to apply. Much like C++, Python supports both procedural and object-oriented programming modes.

Python in recent years has acquired built-in support for functional programming—a set that by most measures includes generators, comprehensions, closures, maps, decorators, anonymous function lambdas, and first-class function objects.

Free: Python is completely free to use and distribute. As with other open source software, such as Tcl, Perl, Linux, and Apache, you can fetch the entire Python system's source code for free on the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python's source code, if you are so inclined.

Portable: The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from PDAs to supercomputers.

Powerful: Python provides all the simplicity and ease of use of a scripting language, along with more advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. Some of the reasons why Python is powerful are: dynamic typing, automatic memory management, built-in object types, etc.

Mixable: Python programs can easily be combined with components written in other languages in a variety of ways. For example, Python's C API lets C programs call and be called by Python programs flexibly. That means you can add functionality to the Python system as needed, and use Python programs within other environments or systems.

Easy to Use: Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps, like there are for languages such as C or C++.

Easy to Learn: When compared to other widely used programming languages, the core Python language is remarkably easy to learn. In fact, if you're an experienced programmer, you can expect to be coding small-scale Python programs in a matter of days, and may be able to pick up some limited portions of the language in just hours.

Python Shell (REPL)

Python's interactive command line or shell, sometimes is called as interactive prompt. The interactive prompt follows REPL (Repeat-Evaluate-Print-Loop). There are a variety of ways to start this command line: in an IDE, from a system console, and so on. Assuming the interpreter is installed as an executable program on your system, the most platform-neutral way to start an interactive interpreter session is usually just to type python at your operating system's prompt, without any arguments.

On Windows, you can type **python** or **py** in a DOS console window - a program named cmd.exe and usually known as Command Prompt. On Linux (and other Unixes), you might type this command in a shell or terminal window.

The Python interactive session begins by printing two lines of informational text giving the Python version number and other information, then prompts for input with >>> when it's waiting for you to type a new Python statement or expression.

When working interactively, the results of your code are displayed below the >>> input lines after you press the Enter key. For instance, here are the results of two Python print statements:

```
% python
>>> print('Hello world!') Hello world!
>>> print(2 ** 8)
256
```

Running Python Scripts

A Python script is a file containing multiple lines of Python code saved with *.py* extension. A Python script can be directly executed line by line without any need to compile it.

Open your favourite text editor (e.g., vi, Notepad, or the IDLE editor), type the following statements into a new text file named script1.py, and save it in your working code directory that you set up earlier:

```
# A first Python script import sys
print(sys.platform)
print(2 ** 100)
x = 'Spam!'
print(x * 8)
```

Once you've saved this text file, you can ask Python to run it by listing its full filename as the first argument to a python command like the following typed at the system shell prompt:

```
% python script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

Variables

Variables are just names of memory locations. Variables are used to store and retrieve values from memory. Type of the variable is based on the value we assign to the variable.

In Python, names come into existence when you assign values to them, but there are a few rules to follow when choosing names for the subjects of your programs:

- Names must start with a letter or underscore and can be followed by any number of letters, digits, or underscores.
- Special symbols are not allowed in names.
- Names are case sensitive.

- Names cannot be reserved words.

Syntax for creating a variable is as follows:

Variable-name = value

Consider the following example which creates a variable named *x* and assigns the value 10 to it:

x = 10

Keywords

Following are keywords available in Python:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

False – False boolean value

True – True boolean value

None – Special constant in Python that represents nothing or a null value

and – Logical and operator

as – Used to create an alias while importing a module

assert – Assert is used to create assertions which are used for debugging

break – Used inside loops to alter its behaviour

class – Used to define a new user-defined class

continue – Used inside loops to alter its behaviour

def – Used to define a used-defined function

del – Used to delete the reference to an object

elif – Else if ladder

else – Used in conjunction with if

except – Used in exception handling

finally – Used in exception handling

for – Is a loop to repeat a set of statements

from – Used to import specific attributes or functions into the current namespace

global – Used to declare a variable inside a function (local variable) as global variable

if – One-way selection statement

import – Used to import modules into the current namespace

in – Used to test if a sequence (string, list, tuple, etc.) contains a value. Also used in a for loop to traverse over a range of values.

is – Used to test object identity (whether two variables are referring to same object or not)

lambda – Used to create an anonymous (nameless) function

nonlocal – Used inside nested functions to say that the variable is modifiable outside the function in which it is being used

not – Logical not operator

or – Logical or operator

pass – Pass is a null statement in Python. Nothing happens when it is executed.

raise – Used in exception handling

return – Used inside a function to exit it and return a value

try – Used in exception handling

while – Is a loop to repeat a set of statements

with - with statement is used to wrap the execution of a block of code within methods defined by the context manager

yield - yield is used inside a function like a return statement. But yield returns a generator.

Input and Output

For accepting input from keyboard, we use *input* function as follows:

input(“String to be displayed...”)

The *input* function returns back a string to the script. To work with numbers, we have to convert the returned string into number as follows:

```
x = int(input("Enter a number: "))
```

To print output to the standard output (terminal window), we use print as follows:

```
print("Message to print...")
```

Some examples of using print function:

```
print("Welcome to Python")
print("Sum of ", x, " and ", y, "is: ", (x+y))
print("hello", end=' ')
print("world")
```

Last two print statements will print "hello world" in a single line.

The *print* functions prints to standard output stream *stdout* which is available in *sys* module. Another way to print output is as follows:

```
import sys
sys.stdout.write("Python Rocks!")
```

Three standard streams are: *stdin, stdout,* and *stderr*.

