4. Python 3 – Basic Syntax

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program

Let us execute the programs in different modes of programming.

Interactive Mode Programming

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

```
$ python
Python 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information.
>>>
On Windows:
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Type the following text at the Python prompt and press Enter-

```
>>> print ("Hello, Python!")
```

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result-

```
Hello, Python!
```

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 When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have the extension **.py**. Type the following source code in a test.py file-

```
print ("Hello, Python!")
```

We assume that you have the Python interpreter set in **PATH** variable. Now, try to run this program as follows-

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result-

```
Hello, Python!
```

Let us try another way to execute a Python script in Linux. Here is the modified test.py file-

```
#!/usr/bin/python3

print ("Hello, Python!")
```

We assume that you have Python interpreter available in the /usr/bin directory. Now, try to run this program as follows-

```
$ chmod +x test.py     # This is to make file executable

$./test.py
```

This produces the following result-

```
Hello, Python!
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers-

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.

- Starting an identifier with a single leading underscore indicates that the identifier is private.

- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

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

Lines and Indentation

Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example-

```
if True:
    print ("True")
else:
  print ("False")
```

However, the following block generates an error-

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
  print ("False")
```

Thus, in Python all the continuous lines indented with the same number of spaces would form a block. The following example has various statement blocks-

Note: Do not try to understand the logic at this point of time. Just make sure you understood the various blocks even if they are without braces.

```
#!/usr/bin/python3
import sys
try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print ("There was an error writing to", file_name)
  sys.exit()
print ("Enter '", file_finish,)
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = input("Enter filename: ")
if len(file_name) == 0:
  print ("Next time please enter something")
```

```
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print ("There was an error reading file")
  sys.exit()
file_text = file.read()
file.close()
print (file_text)
```

Multi-Line Statements

Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue. For example-

```
total = item_one + \
        item_two + \
        item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example-

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple ('' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal-

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python3
```

```
# First comment
print ("Hello, Python!") # second comment
```

This produces the following result-

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression-

```
name = "Madisetti" # This is again comment
```

Python does not have multiple-line commenting feature. You have to comment each line individually as follows-

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt and the statement saying "Press the enter key to exit", and then waits for the user to take action –

```
#!/usr/bin/python3
input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block. Here is a sample snip using the semicolon-

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

Groups of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with **-h**:

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit
[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advance topic. Let us understand it.

Command Line Arguments

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
$ python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes-

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here sys.argv[0] is the program i.e. the script name.

Example

Consider the following script **test.py-**

```
#!/usr/bin/python3
import sys
print ('Number of arguments:', len(sys.argv), 'arguments.')
print ('Argument List:', str(sys.argv))
```

Now run the above script as follows –

```
$ python test.py arg1 arg2 arg3
```

This produces the following result-

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

NOTE: As mentioned above, the first argument is always the script name and it is also being counted in number of arguments.

Parsing Command-Line Arguments

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

getopt.getopt method

This method parses the command line options and parameter list. Following is a simple syntax for this method-

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters-

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns a value consisting of two elements- the first is a list of **(option, value)** pairs, the second is a list of program arguments left after the option list was stripped.

- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

Exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

Example

Suppose we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows-

```
usage: test.py -i <inputfile> -o <outputfile>
```

Here is the following script to test.py-

```
#!/usr/bin/python3
import sys, getopt
def main(argv):
   inputfile = ''
   outputfile = ''
   try:
      opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
   except getopt.GetoptError:
      print ('test.py -i <inputfile> -o <outputfile>')
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print ('test.py -i <inputfile> -o <outputfile>')
         sys.exit()
      elif opt in ("-i", "--ifile"):
         inputfile = arg
      elif opt in ("-o", "--ofile"):
         outputfile = arg
   print ('Input file is "', inputfile)
   print ('Output file is "', outputfile)
if __name__ == "__main__":
   main(sys.argv[1:])
```

Now, run the above script as follows-

```
$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile -o outputfile
Input file is " inputfile
Output file is " outputfile
```