

UNIT-6

PYTHON

1. Python: Introduction to Python language

Python is a general purpose, dynamic, high level, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Python is *easy to learn* yet powerful and versatile scripting language, which makes it attractive for Application Development.

Python's syntax and *dynamic typing* with its interpreted nature make it an ideal language for scripting and rapid application development.

Python supports *multiple programming pattern*, including object-oriented, imperative, and functional or procedural programming styles.

Python is not intended to work in a particular area, such as web programming. That is why it is known as *multipurpose* programming language because it can be used with web, enterprise, 3D CAD, etc.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write `a=10` to assign an integer value in an integer variable.

Python makes the development and debugging *fast* because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

Python 2 vs. Python 3

In most of the programming languages, whenever a new version releases, it supports the features and syntax of the existing version of the language, therefore, it is easier for the projects to switch in the newer version. However, in the case of Python, the two versions Python 2 and Python 3 are very much different from each other.

A list of differences between Python 2 and Python 3 are given below:

1. Python 2 uses `print` as a statement and used as `print "something"` to print some string on the console. On the other hand, Python 3 uses `print` as a function and used as `print("something")` to print something on the console.

2. Python 2 uses the function `raw_input()` to accept the user's input. It returns the string representing the value, which is typed by the user. To convert it into the integer, we need to use the `int()` function in Python. On the other hand, Python 3 uses `input()` function which automatically interpreted the type of input entered by the user. However, we can cast this value to any type by using primitive functions (`int()`, `str()`, etc.).
3. In Python 2, the implicit string type is ASCII, whereas, in Python 3, the implicit string type is Unicode.
4. Python 3 doesn't contain the `xrange()` function of Python 2. The `xrange()` is the variant of `range()` function which returns a xrange object that works similar to Java iterator. The `range()` returns a list for example the function `range(0,3)` contains 0, 1, 2.
5. There is also a small change made in Exception handling in Python 3. It defines a keyword `as which` is necessary to be used. We will discuss it in Exception handling section of Python programming tutorial.

2. Python-syntax

Python provides us the two ways to run a program:

- Using Interactive interpreter prompt
- Using a script file

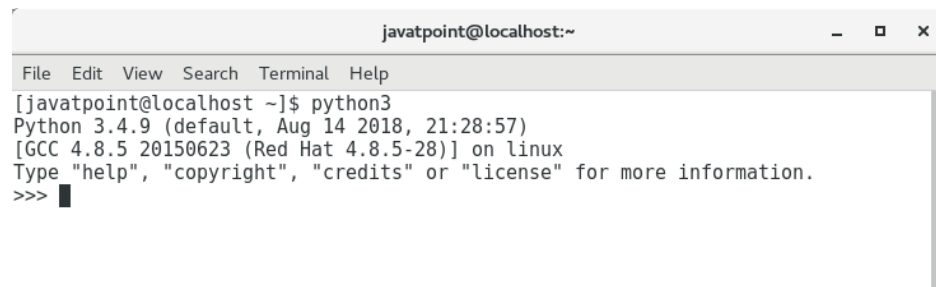
Interactive interpreter prompt

Python provides us the feature to execute the python statement one by one at the interactive prompt. It is preferable in the case where we are concerned about the output of each line of our python program.

To open the interactive mode, open the terminal (or command prompt) and type `python` (`python3` in case if you have `python2` and `python3` both installed on your system).

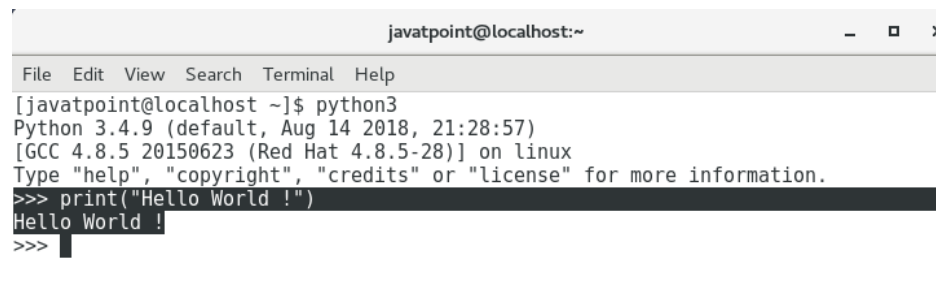
It will open the following prompt where we can execute the python statement and

check their impact on the console.



```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python3  
Python 3.4.9 (default, Aug 14 2018, 21:28:57)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

Let's run a python statement to print the traditional hello world on the console. Python3 provides `print()` function to print some message on the console. We can pass the message as a string into this function. Consider the following image.



```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python3  
Python 3.4.9 (default, Aug 14 2018, 21:28:57)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello World !")  
Hello World !  
>>> █
```

Here, we get the message **"Hello World !"** printed on the console.

Using a script file

Interpreter prompt is good to run the individual statements of the code. However, we cannot write the code every-time on the terminal.

We need to write our code into a file which can be executed later. For this purpose, open an editor like notepad, create a file named `first.py` (python used `.py` extension) and write the following code in it.

Print ("`hello world`"); #here, we have used `print()` function to print the message on the console.

To run this file named as `first.py`, we need to run the following command on the terminal.

\$ `python3 first.py`

```
javatpoint@loca
File Edit View Search Terminal Tabs Help
javatpoint@localhost:~ x
[javatpoint@localhost ~]$ python3 first.py
hello world !
[javatpoint@localhost ~]$ █
```

Hence, we get our output as the message **Hello World !** is printed on the console.

3. Statements:

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 which will be discussed later.

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:

1. `a = 1 + 2 + 3 + \`
2. `4 + 5 + 6 + \`
3. `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

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

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

1. `colors = ['red',`
2. `'blue',`

3. 'green']

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

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

4. Functions

In Python, function is a group of related statements that perform a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes code reusable.

Syntax of Function

def function_name(parameters):

"""docstring"""

statement(s)

Above shown is a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example of a function

def greet(name):

"""This function greets to
 the person passed in as

```
parameter"""
```

```
print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
```

```
Hello, Paul. Good morning!
```

5. Built-in-functions and Methods

Python Built-in Function

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are 68 built-in functions. They are listed below alphabetically along with brief description.

Method	Description
Python abs()	returns absolute value of a number
Python all()	returns true when all elements in iterable is true
Python any()	Checks if any Element of an Iterable is True
Python ascii()	Returns String Containing Printable Representation
Python bin()	converts integer to binary string

<u>Python bool()</u>	Converts a Value to Boolean
<u>Python bytearray()</u>	returns array of given byte size
<u>Python bytes()</u>	returns immutable bytes object
<u>Python callable()</u>	Checks if the Object is Callable
<u>Python chr()</u>	Returns a Character (a string) from an Integer
<u>Python classmethod()</u>	returns class method for given function
<u>Python compile()</u>	Returns a Python code object
<u>Python complex()</u>	Creates a Complex Number
<u>Python delattr()</u>	Deletes Attribute From the Object
<u>Python dict()</u>	Creates a Dictionary
<u>Python dir()</u>	Tries to Return Attributes of Object
<u>Python divmod()</u>	Returns a Tuple of Quotient and Remainder
<u>Python enumerate()</u>	Returns an Enumerate Object
<u>Python eval()</u>	Runs Python Code Within Program
<u>Python exec()</u>	Executes Dynamically Created Program
<u>Python filter()</u>	constructs iterator from elements which are true

<u>Python float()</u>	returns floating point number from number, string
<u>Python format()</u>	returns formatted representation of a value
<u>Python frozenset()</u>	returns immutable frozenset object
<u>Python getattr()</u>	returns value of named attribute of an object
<u>Python globals()</u>	returns dictionary of current global symbol table
<u>Python hasattr()</u>	returns whether object has named attribute
<u>Python hash()</u>	returns hash value of an object
<u>Python help()</u>	Invokes the built-in Help System
<u>Python hex()</u>	Converts to Integer to Hexadecimal
<u>Python id()</u>	Returns Identify of an Object
<u>Python input()</u>	reads and returns a line of string
<u>Python int()</u>	returns integer from a number or string
<u>Python isinstance()</u>	Checks if a Object is an Instance of Class
<u>Python isinstance()</u>	Checks if a Object is Subclass of a Class
<u>Python iter()</u>	returns iterator for an object
<u>Python len()</u>	Returns Length of an Object

<u>Python list() Function</u>	creates list in Python
<u>Python locals()</u>	Returns dictionary of a current local symbol table
<u>Python map()</u>	Applies Function and Returns a List
<u>Python max()</u>	returns largest element
<u>Python memoryview()</u>	returns memory view of an argument
<u>Python min()</u>	returns smallest element
<u>Python next()</u>	Retrieves Next Element from Iterator
<u>Python object()</u>	Creates a Featureless Object
<u>Python oct()</u>	converts integer to octal
<u>Python open()</u>	Returns a File object
<u>Python ord()</u>	returns Unicode code point for Unicode character
<u>Python pow()</u>	returns x to the power of y
<u>Python print()</u>	Prints the Given Object
<u>Python property()</u>	returns a property attribute
<u>Python range()</u>	return sequence of integers between start

	and stop	
Python repr()	returns printable representation of an object	
Python reversed()	returns reversed iterator of a sequence	
Python round()	rounds a floating point number to ndigits places.	
Python set()	returns a Python set	
Python setattr()	sets value of an attribute of object	
Python slice()	creates a slice object specified by range()	
Python sorted()	returns sorted list from a given iterable	
Python staticmethod()	creates static method from a function	
Python str()	returns informal representation of an object	
Python sum()	Add items of an Iterable	
Python super()	Allow you to Refer Parent Class by super	
Python tuple() Function	Creates a Tuple	
Python type()	Returns Type of an Object	
Python vars()	Returns __dict__ attribute of a class	
Python zip()	Returns an Iterator of Tuples	

5. Modules in python

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

```
1. # Python Module example
2.
3. def add(a, b):
4.     """This program adds two
5.     numbers and return the result"""
6.
7.     result = a + b
8.     return result
```

Here, we have defined a [function](#) add() inside a module named example. The function takes in two numbers and returns their sum.

How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

```
>>> import example
```

This does not enter the names of the functions defined in example directly in the current symbol table. It only enters the module name example there.

Using the module name we can access the function using the dot .operator. For example:

```
>>> example.add(4,5.5)
```

```
9.5
```

Python has a ton of standard modules available.

You can check out the full list of [Python standard modules](#) and what they are for. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed as follows.

Python import statement

We can import a module using import statement and access the definitions inside it using the dot operator as described above. Here is an example.

1. # import statement example
2. # to import standard module math
3. import math
4. print("The value of pi is", math.pi)

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

6. Exception Handling

What are exceptions in Python?

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and

then to A.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.

A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

```
1. # import module sys to get the type of exception
2. import sys
3. randomList = ['a', 0, 2]
4. for entry in randomList:
5.     try:
6.         print("The entry is", entry)
7.         r = 1/int(entry)
8.         break
9.     except:
10.        print("Oops!",sys.exc_info()[0],"occured.")
11.        print("Next entry.")
12.        print()
13.        print("The reciprocal of",entry,"is",r)
```

Output

The entry is a

Oops! <class 'ValueError'> occured.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError' > occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
1. try:
2.     # do something
3.     pass
4.
5. except ValueError:
6.     # handle ValueError exception
7.     pass
8.
9. except (TypeError, ZeroDivisionError):
10.    # handle multiple exceptions
11.    # TypeError and ZeroDivisionError
12.    pass
13.
14. except:
15.    # handle all other exceptions
16.    pass
```

Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword `raise`.

We can also optionally pass in value to the exception to clarify why that exception was raised.

1. `>>> raise KeyboardInterrupt`
2. Traceback (most recent call last):
3. ...
4. `KeyboardInterrupt`
- 5.
6. `>>> raise MemoryError("This is an argument")`
7. Traceback (most recent call last):
8. ...
9. `MemoryError: This is an argument`
- 10.
11. `>>> try:`
12. `... a = int(input("Enter a positive integer: "))`
13. `... if a <= 0:`
14. `... raise ValueError("That is not a positive number!")`
15. `... except ValueError as ve:`
16. `... print(ve)`
17. `...`
18. Enter a positive integer: -2
19. That is not a positive number!

try...finally

The try statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the `finally` clause to guarantee execution.

Here is an example of [file operations](#) to illustrate this.

```
1. try:
2.     f = open("test.txt",encoding = 'utf-8')
3.     # perform file operations
4. finally:
5.     f.close()
```

This type of construct makes sure the file is closed even if an exception occurs.

IMPORTANT QUESTIONS:

- 1 Explain how python is compared with other programming languages?
- 2 Discuss the sequence types that python supports. Explain in detail about strings?
- 3 Explain in detail about lists and tuples?
- 4 Discuss about mutable and immutable types of objects?
- 5 Explain simple statements in Python?
- 6 Explain the working of for and while loop?
- 7 Describe the concept of functions with an example?
- 8 Explain the features of Python language?

- 9 Explain built-in functions for numeric types?
- 10 Discuss about built-in functions and methods for files?
- 11 Write a program that asks the user for a weight in kilograms and converts it to pounds. There are 2.2 pounds in a kilogram.
- 12 Write a program to fill the screen horizontally and vertically with your name. [Hint: add the option end=" into the print function to fill the screen horizontally.]
- 13 Explain Exception Handling mechanisms in Python.
- 14 Describe Web application framework using Python.
- 15 Write a program that uses a for loop to print the numbers 100, 98, 96, . . 4, 2.?