Learning Report – Advanced Python Programming

L&T Technology Services

Document History

Ver. Rel. No.	Release Date	Prepared. By	Reviewed By	To be approved By	Remarks/Revision Details
1	07/02/2021	Sneha Anand			

Table of Contents

FUNCTIONS 4
CLASSES IN PYTHON 5
LISTS 6
TUPLES 7
SETS 8
INHERITANCE IN PYTHON 9
VARIABLES IN PYTHON 14
EXCEPTIONS IN PYTHON 15
REGULAR EXPRESSIONS IN PYTHON 17

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. Functions in Python are first-class citizens. It means that functions have equal status with other objects in Python. Functions can be assigned to variables, stored in collections, or passed as arguments. This brings additional flexibility to the language.

- Functions in python is an aggregation of related statement designed to perform a complex task.
- Functions can be inbuilt or user defined according to the user needs.
- Functions helps to reduce the code complexity and code length.

ADVANTAGES OF FUNCTIONS IN PYTHON:

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

PYTHON FUNCTION TYPES:

There are two basic types of functions: built-in functions and user defined functions. The built-in functions are part of the Python language; for instance dir(), len(), or abs(). The user defined functions are functions created with the def keyword.

EXAMPLE FOR PYTHON FUCTIONS:

```
def absolute _value(num):
        ```This function returns the absolute value of the entered number```
        If num >= 0:
                return num
        Else:
                return -num

print(absolute_value(2))
print(absolute_value(-4))
```

Output:

2
4
```

CLASSES IN PYTHON

Like function definitions begins with the def keyword in python, class definitions begin with a class keyword. A class is a user-defined function in which objects are created. Creation of a new class creates a new object.A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instance can have attributes attached to it for maintaining its state. Class instance can also have methods (defined by their class) for modifying their state.

CLASSES SYNTAX:

```
Class ClassName:

    # Statement-1
    .
    .
    .
    #Statement-n
```

ADVANTAGES OF CLASS IN PYTHON:

- Classes are useful when a collection of attributes and methods are being used repeatedly.
- A class can be made public or private according to the user.

EXAMPLE FOR PYTHON CLASS:

```
Class person:
def __init__(self, name, age):
self.name = name
self.age = age

p1 = person("John",23)
print(p1.name)
print(p1.age)
```

OUTPUT:

```
John
23
```

LISTS

A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets.
Lists are great to use when you want to work with many related values. They enable you to keep data together that belongs together, condense your code, and perform the same methods and operations on multiple values at once.

LIST SYNTAX:

```
L1 = [“apple”, “banana”, “cherry”]
Print(L1)
```

OUTPUT:

```
[‘apple’, ‘banana’, ‘cherry’]
```

ADVANTAGES OF LISTS IN PYTHON:

- The code is more concise.
- The code is generally more readable.
- The code, in most cases, will run faster.

EXAMPLE FOR PYTHON LISTS:

```
L1 = [‘physics’, ‘chemistry’, 1998, 2000]
Print L1
del L1[2]
print L1
```

OUTPUT:

```
[‘physics’, ‘chemistry’, 1998, 2000]
[‘physics’, ‘chemistry’, 2000]
```

TUPLES

Tuple is a collection of python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers. Values of a tuple are syntactically separated by 'commas' with or without the use of parentheses for grouping the data sequence.

TUPLE SYNTAX:

```
Thistuple = ("apple", "banana", "cherry")
Print(thistuple)
```

OUTPUT:

("apple", "banana", "cherry")

ADVANTAGES OF LISTS IN PYTHON:

- Tuples are fined size in nature i.e.we can't add/delete elements to/from a tuple.
- We can search any element in a tuple.
- Tuples are faster than lists, because they have a constant set of values.
- Tuples can be used as dictionary keys, because they contain immutable values like strings, numbers, etc.

EXAMPLE FOR PYTHON TUPLE:

```
#creating an empty tuple
Tuple1 = ()
print ("Empty Tuple: ")
print (Tuple1)

#creating an tuple with use of string
Tuple1 = ('ltts', 'intern')
print ("\nTuple with string: ")
print (Tuple1)

#creating an tuple with use of list
list1 = (1, 2, 3, 4, 5)
print ("\nTuple with list: ")

print (Tuple(list1)
```

OUTPUT:

```
Empty Tuple:
()
Tuple with string:
('ltts', 'intern')
Tuple with list:
(1, 2, 3, 4, 5)
```

SETS

A Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. Since sets are unordered, we cannot access items using indexes like we do in lists.

SETS SYNTAX:

```
my_set = set(["a", "b", "c"])
print(my_set)
```

OUTPUT:

```
{'c', 'b', 'a'}
```

ADVANTAGES OF SETS IN PYTHON:

Because sets cannot have multiple occurrences of the same element, it makes sets highlyuseful to efficiently remove duplicate values from a list or tuple and to perform common math operations like unions and intersections.

EXAMPLE FOR PYTHON SETS:

```
Days = set([“mon”,”tue”, “wed”, “thu”, “fri”, “sat”, “sun”])
Months = {“Jan”, “feb”, “mar”}
Dates = {20, 25, 30}
Print (Days)
Print (Months)
Print (Dates)
```

OUTPUT:

```
set([“mon”,”tue”, “wed”, “thu”, “fri”, “sat”, “sun”])
set([“Jan”, “feb”, “mar”])
set([20, 25, 30])
```

INHERITANCE IN PYTHON

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

DIFFERENT FORMS OF INHERITANCE:

- **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

- **Multiple inheritance**: When a child class inherits from multiple parent classes, it is called multiple inheritance.
  Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as a comma-separated list in the bracket.

- **Multilevel inheritance**: When we have a child and grandchild relationship.

- **Hierarchical inheritance** More than one derived classes are created from a single base.

- **Hybrid inheritance**: This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

SETS SYNTAX:

```
class BaseClass:
        Body of base class
class DerivedClass (BaseClass):
        Body of derived class
```

EXAMPLE FOR PYTHON SINGLE INHERITANCE:

```
class Parent:
        Def func1(self):
                Print("This function is in parent class.")
class Child (Parent):
        Def func2(self):
                Print("This function is in child class.")
object = Child()
object.func1()
object.func2()
```

OUTPUT:

```
This function is in parent class.
This function is in child class.
```

EXAMPLE FOR PYTHON MULTIPLE INHERITANCE:

```
class Mother:
mothername = ""
def mother(self):
        print(self.mothername)

class Father:
fathername = ""
def father(self):
        print(self.fathername)

class Son(Mother, Father):
def parents(self):
         print("Father :", self.fathername)
         print("Mother :", self.mothername)

s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

OUTPUT:

```
Father : Rahul

Mother : Saritha
```

EXAMPLE FOR PYTHON MULTILEVEL INHERITANCE:

```
class Grandfather:
def __init__(self, grandfathername):
self.grandfathername = grandfathername

class Father(Grandfather):
def __init__(self, fathername, grandfathername):
self.fathername = fathername

# invoking constructor of Grandfather class
Grandfather.__init__(self, grandfathername)

class Son(Father):
def __init__(self,sonname, fathername, grandfathername):
self.sonname = sonname

# invoking constructor of Father class
 Father.__init__(self, fathername, grandfathername)

def print_name(self):
print('Grandfather name :', self.grandfathername)
print("Father name :", self.fathername)
 print("Son name :", self.sonname)

s1 = Son('Rahul', 'Ram', 'Rehan')
print(s1.grandfathername)
s1.printname()
```

OUTPUT:

```
Lal mani

Grandfather name :Rehan

Father name : Ram

Son name : Rahul
```

EXAMPLE FOR PYTHON HIERARCHICAL INHERITANCE:

```
class Parent:
def func1(self):
        print("This function is in parent class.")

class Child1(Parent):
def func2(self):
        print("This function is in child 1.")

class Child2(Parent):
def func3(self):
        print("This function is in child 2.")

object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

OUTPUT:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

EXAMPLE FOR PYTHON HYBRID INHERITANCE:

```
class School:
def func1(self):
        print("This function is in school.")

class Student1(School):
 def func2(self):
         print("This function is in student 1. ")

class Student2(School):
def func3(self):
        print("This function is in student 2.")

class Student3(Student1, School):
def func4(self):
        print("This function is in student 3.")

object = Student3()
object.func1()
object.func2()
```

OUTPUT:

```
This function is in school.

This function is in student 1
```

VARIABLES IN PYTHON

In many programming languages, variables are statically typed. That means a variable is initially declared to have a specific data type, and any value assigned to it during its lifetime must always have that type. Variables in Python are not subject to this restriction. In Python, a variable may be assigned a value of one type and then later re-assigned a value of a different type.

EXAMPLE FOR PYTHON MULTILEVEL INHERITANCE:

```
var = 23.5

print (var)

var = “now I am a string”

print (var)
```

OUTPUT:

```
23.5
Now I am a string
```

EXCEPTIONS IN PYTHON

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.

In Python, exceptions can be handled using a try statement. The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause. We can thus choose what operations to perform once we have caught the exception.

EXAMPLE FOR EXCEPTIONS IN PYTHON

```
    Import sys
    randomList = ['a', 0, 2]
    for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except :
        print("oops!" , sys.exc_info()[0], "occurred.")
        print("next entry")
        print()
print("The reciprocal of", entry, "is", r)
```

OUTPUT:

The entry is a

Oops! <class 'valueerror'>occurred

Next entry

The entry is 0

Oops! <class 'zerodivisionerror'>occurred

Next entry

The entry is 2

The reciprocal of 2 is 0.5.

REGULAR EXPRESSIONS IN PYTHON

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as r'expression'.

Python offers two different primitive operations based on regular expressions:

match checks for a match only at the beginning of the string.

search checks for a match anywhere in the string.

EXAMPLE FOR REGULAR EXPRESSIONS IN PYTHON

➔ Import re

Print(re.match(“hello”,”hello”))

Match hello

➔ Import re

Print(re.match(“hello”,”hi hello”))

Not Match hello

➔ Import re

Print(re.search(“hello”,”hi hello”))

Match hello

? (0 or 1) preceding:

➔ Import re

Print(re.match(“he?llo”,”hllo”))
Match hllo

➔ Import re

Print(re.match(“he?llo”,”hello”))
Match hello

➔ Import re

Print(re.match(“he?llo”,”heello”))
Not Match hello

+ (1 or more times) preceding:

➔ Import re

Print(re.match(“he+llo”,”hello”))
Match hello

➔ Import re

Print(re.match(“he+llo”,”heello”))
Match hello

➔ Import re

Print(re.match(“he+llo”,”hllo”))
Not Match hello

*** (0 or more times) preceding**:

➔ Import re

Print(re.match("he*llo","hello"))
Match hello

➔ Import re

Print(re.match("he*llo","hllo"))
Match hllo

➔ Import re

Print(re.match("he*llo","heello"))
Match hello

[]-match any single character in the list:

➔ Import re

Print(re.match("[abc]","abc123"))
Match a

➔ Import re

Print(re.match("[abc]+","abc123"))
Match abc

➔ Import re

Print(re.match("[0-9]+","bc123"))
Not match

[^] opposite words there than it will match:

➔ Import re

Print(re.match("[^abc]+","123abc"))
Match 123

➔ Import re

Print(re.match("[^abc]+","abc123"))
Not Match

➔ Import re

Print(re.search("[^abc]+","123abc"))
Match 123

(|) either or condition:

➔ Import re

Print(re.match("[a-z]+|[0-9]+","abc123"))
Match abc

Groupin():

➔ Import re

Print(re.match("(ab)+c","ababc"))
Match ababc

➔ Import re

Print(re.search("(ab)c","ababc"))
Match abc

^-Start of the string:

➔ Import re

Print(re.match("^abc","abc"))
Match abc

➔ Import re

Print(re.match("^abc","abd"))
Not Match

$-End of the string:

➔ Import re

Print(re.match("^abc$","abc"))
Match abc

➔ Import re

Print(re.match("^abc$","abcd"))
Not Match

{M,N}-Minimum and Maximum:

➔ Import re

Print(re.match("[a-z]{2,3}","abcdef"))
Match abcde