upGrad

SUMMARY

Introduction to Python

Table of Contents

What is Python?

Python is a high-level scripting language which can be used for various text processing, system administration and internet-related tasks. Python is a true object-oriented language and is available on a wide variety of platforms.

upGrad

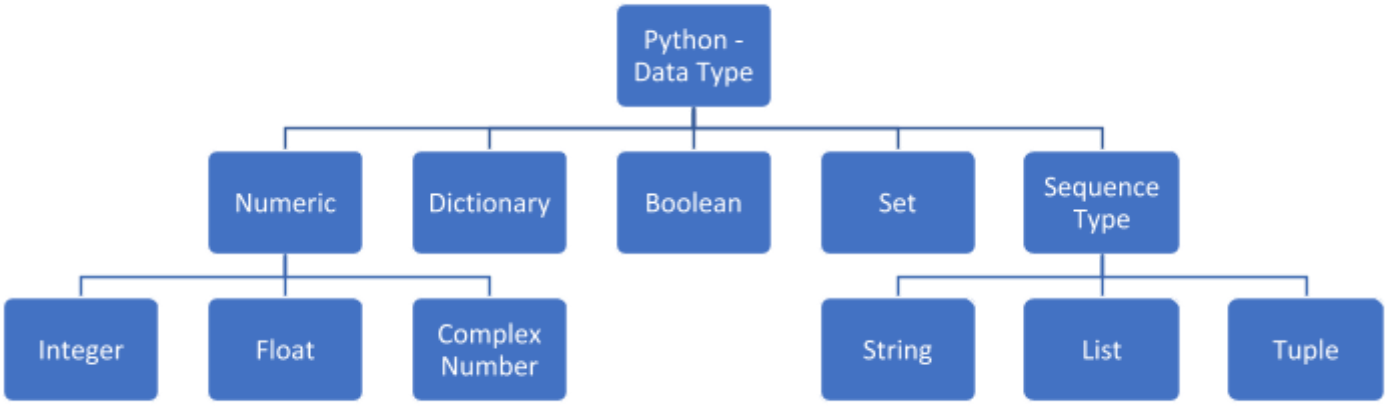
Basics of Python

Following are a few features of Python which differentiate it from other programming languages

- ✔ Python statements do not need to end with a special character – the Python interpreter knows that you are done with an individual statement with the presence of a newline which gets generated when you press the 'Return' key of the keyboard.
- ✔ If a statement spans more than one line, the safest course of action is to use a backslash (\) at the end of the line to let Python know that you are going to continue the statement on the next line.
- ✔ Python uses indentation, that is, the amount of white space before the statement itself, to indicate the presence of loops.
- ✔ Object Oriented Programming: Python is a true object-oriented language. The basic concept of object oriented programming is encapsulation, which is the ability to define an object that contains the data and all the information a program needs to operate on that data.

Data Types

Python data types can be classified into the following categories.



upGrad

String Data

Strings are a collection of characters. You can create a string constant inside a Python program by enclosing the text with single quotes (’), double quotes ("), or a collection of three different types of quotes (’’’ or """).

```
a = "Hello"
print(a)
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

```
Hello
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

String Operations

- Concatenation

(+) takes on the role of concatenation of strings, which is a simple addition of strings.

```
first = "Foot"
second = "Ball"
print(first + second)
```

```
FootBall
```

You can freely mix variables and string constants together — anywhere that Python expects a string, it can be either a variable or a constant.

```
str = """Send me email
My address is """
print(str + "test@upgrad.com")
```

```
Send me email
My address is test@upgrad.com
```

Note:
Python checks the type of variable before carrying out operations:
x = “12.”/7.
Print(’The answer is ’ + x)
Traceback (innermost last):File "<stdin>", line 1, in ?TypeError: illegal argument type for built-in operation

Repetition

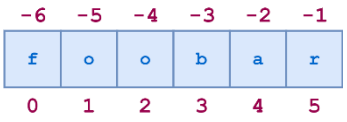
The asterisk (*) creates a new string with the string repeated by the value of the number.

```
str = """Send me email
My address is """
print(str + "test@upgrad.com")
```

```
Send me email
My address is test@upgrad.com
```

Indexing and Slicing

To extract a character from a string, use a string with square brackets ([ ]), the first character of a string has index **0**. If the subscript is less than zero, Python counts from the end of the string, with a subscript of -1 representing the last character in the string. As in the following example, the index starts from 0 on the left and -1 on the right.




```
str = """Send me email
My address is """
print(str + "test@upgrad.com")
```

```
Send me email
My address is test@upgrad.com
```

```
str = 'This carrot is red'
# printing the third character
print(str[3])
# printing the first character
print(str[0])
# printing the character using the negative index
print(str[-1])
```

```
s
T
d
```

To get a slice, use a subscript by providing the starting position followed by a colon (:), followed by one more than the ending position of the slice you want to extract. Notice that the slicing stops immediately before the second value. Following is an example.

```
# Slicing the string
print(str[0:4])
```

```
This
```

```
print(str[5:11])
```

```
carrot
```

upGrad

Functions and Methods for Character Strings

Method	Description
capitalize()	Converts the first character to uppercase
casefold()	Converts string into lowercase
center()	Returns a centred string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lowercase becomes uppercase and vice versa
title()	Converts the first character of each word to uppercase
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Refer : https://www.w3schools.com/python/python_ref_string.asp

upGrad

Numeric Data

Integer - int

Int holds signed integers. You can use the type() function to find which class it belongs to.

```
a=-7
a=99999999999999999999999999999999
print(type(a))
```

```
<class 'int'>
```

Length of the integer is dependent on the memory available.

Float

Float holds floating point real values.

```
b=3.566
print(type(b))
```

```
<class 'float'>
```

Complex

Complex data type holds a complex number. Complex numbers are of the format: a+bi. Here, a and b are the real parts of the number and i is imaginary.

```
z = complex(2, -3)
print(z)
print(type(z))
```

```
(2-3j)
<class 'complex'>
```

Lists

A list is a collection of values. List may contain different types of values. The list can be defined as follows.

```
days=['Monday','Tuesday',3,4,5,6,7]
mixList  =  [7,'dog','tree',[1,5,2,7],'abs']
print(days)
print(mixList)
```

```
['Monday', 'Tuesday', 3, 4, 5, 6, 7]
[7, 'dog', 'tree', [1, 5, 2, 7], 'abs']
```

Operations on List

- Slicing a List: Split the list in parts.

```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
sliced_nums = nums[2:7]
print(sliced_nums)
```

```
[30, 40, 50, 60, 70]
```

Indexing for a list starts with 0, like for a string. Just like a string, negative indexing means starting from the last element of the list.

- Len(): Length of a list is an inbuilt function.

```
print(len(nums))
```

```
9
```

- A list is mutable, that is, reassign elements.

```
num = [10, 20, 30, 40, 50, 60, 70, 80, 90]
num[0] = 0
print(num )
```

```
[0, 20, 30, 40, 50, 60, 70, 80, 90]
```

- Accessing elements of the List means that to loop over the list, we can use the **for** loop.

```
num = [1,2,5,6,8]
for n in num:
    print(n)
```

```
1
2
5
6
8
```

- Multidimensional Lists: A list may have more than one dimension.

```
#Multidimensional Lists- A list may have more than one dimension.
a=[[1,2,3],[4,5,6]]
print(a)
nestlist  =  [1,2,[10,20,30,[7,9,11,[100,200,300]]],[1,7,8]]
print(nestlist)
```

```
[[1, 2, 3], [4, 5, 6]]
[1, 2, [10, 20, 30, [7, 9, 11, [100, 200, 300]]], [1, 7, 8]]
```

- Concatenation: Use a plus sign (+) between the two lists to be concatenated.

```
#Concatenation: To combine the contents of two lists, use a plus sign (+) between the two lists to be concatenated.
first  =  [7,9,'dog']
second = ['cat',13,14,12]
print(first + second)
```

```
[7, 9, 'dog', 'cat', 13, 14, 12]
```

- Repetition of List

```
#Repetition of List
print(['a','b','c'] * 4)
```

```
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

- Extend Append List

The extend() function works only on the list. However, you can append() an integer to the list.

```
#Extend  And Append List

a = [1,5,7,9]
b  =  [10,20,30,40]
a.extend(b)
print(a)
```

```
[1, 5, 7, 9, 10, 20, 30, 40]
```

```
 a = [1,5,7,9]
 a.append(b)
print( a)
```

```
[1, 5, 7, 9, [10, 20, 30, 40]]
```

```
 a = [1,5,7,9]
b= 4
a.append(b)
print(a)
```

```
[1, 5, 7, 9, 4]
```

Tuples

A tuple is like a list. It differs from the list in using parentheses instead of square brackets.

```
####Tuples
## A tuple is like a list. It differs from the list in using parentheses instead of square brackets.
```

```
fruit=('Apple','Banana','Orange')
print(fruit)
```

```
('Apple', 'Banana', 'Orange')
```

Operation on Tuples

- Accessing and Slicing a Tuple – It is the same as a list.

```
fruit=('Apple','Banana','Orange')
print(fruit[1])
```

```
Banana
```

```
print(fruit[0:2])
```

```
('Apple', 'Banana')
```

Python Tuple is immutable. Once declared, you cannot change its size or elements.
fruit [2]='Mango'
Traceback (most recent call last):
File "<pyshell#107>", line 1, in <module>
fruit [2]='Mango'
TypeError: 'tuple' object does not support item assignment

Dictionaries

A dictionary is a key-value pair. It can be defined as follows.

```
person={'city':'Ahmedabad','age':7}
print(person)
```

```
{'city': 'Ahmedabad', 'age': 7}
```

- Accessing a Value: To access a value, use the key in square brackets.

```
#Accessing a Value- To access a value, you use key in square brackets.

print(person['city'])
```

```
Ahmedabad
```

- Reassigning Elements

```
#Reassigning Elements
person['age']=21
print(person['age'])
```

```
21
```

- List of Keys

```
phonedict  =  {'Fred':'555-1231','Andy':'555-1195','Sue':'555-2193'}
print(phonedict.keys() )
```

```
dict_keys(['Fred', 'Andy', 'Sue'])
```

```
print(phonedict.values())
```

```
dict_values(['555-1231', '555-1195', '555-2193'])
```

Boolean

A Boolean can take true or false values.

```
a=2>1
print(a)
```

```
True
```

Sets

A set can have a list of values defined using {}.

```
a={1,2,3}
print(a)
```

```
{1, 2, 3}
```

- The add() and remove() functions can be used for adding and removing elements. Every set element is unique without duplicates and must be immutable, that is, it cannot be changed. However, a set itself is mutable.

```
#add() and remove() for adding and removing element. Set is also immutable.

a={1,2,3,4}
print(a)
a.remove(4)
print(a)
a.add(5)
print(a)
```

```
{1, 2, 3, 4}
{1, 2, 3}
{1, 2, 3, 5}
```

upGrad

Arithmetic Operations

Operation	Result	Notes	Full documentation
x + y	Sum of *x* and *y*		
x - y	Difference of *x* and *y*		
x * y	Product of *x* and *y*		
x / y	Quotient of *x* and *y*		
x // y	Floored quotient of *x* and *y*	(1)	
x % y	Remainder of x / y	(2)	
-x	*x* negated		
+x	*x* unchanged		
abs(x)	Absolute value or magnitude of *x*		abs()
int(x)	*x* converted to integer	(3)(6)	int()
float(x)	*x* converted to floating point	(4)(6)	float()
complex(re, im)	A complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	(6)	complex()
c.conjugate()	Conjugate of the complex number *c*		
divmod(x, y)	The pair (x // y, x % y)	(2)	divmod()
pow(x, y)	*x* to the power *y*	(5)	pow()
x ** y	*x* to the power *y*	(5)	

Integer Division: The result value is a whole integer. The result is always rounded towards minus infinity.
1//2 is 0,
(-1)//2 is -1,
1//(-2) is -1, and
(-1)//(-2) is 0.
Python defines pow(0, 0) and 0 ** 0 to be 1, as is common for programming languages.

If-else statements

The basic form of the statement can be given as:

```
if expression:
        statement(s)
elif expression:
         statement(s)
elif expression:
        statement(s)
. . .
else:
        statements
```

```
x=2
if  x  ==  1:
    z = 1
    print("Setting  z  to  1")
elif  x  ==  2:
    y = 2
    print("Setting  y  to  2")
elif  x  ==  3:
    w = 3
    print( "Setting  w  to  3")
```

```
Setting  y  to  2
```

Loops

for loops

The basic form of the for loop is as follows:

for var in sequence:
    statements

```
names  =  [('Smith','John'),('Jones','Fred'),('Williams','Sue')]
for i in names:
    print('s',i[1], i[0])
```

```
s John Smith
s Fred Jones
s Sue Williams
```

- The for loops and range functions: The range function accepts one, two or three arguments. With a single integer argument, the range returns a sequence of integers from 0 to one less than the argument provided. With two arguments, the first argument is used as a starting value instead of 0, and with three arguments, the third argument is used as an increment instead of the implicit default of 1.

```
prices  =  [12.00,14.00,17.00]
taxes = [0.48,0.56,0.68]
total = []
for  i  in  range(len(prices)):
    total.append(prices[i]  +  taxes[i])

print(total)
```

```
[12.48, 14.56, 17.68]
```

while loops

The basic syntax of the while loop is as follows:

while expression:
    statements
else:

statements

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

Map, Filter and Reduce

Map

Map applies a function to all the items in an input list. The basic syntax of the map function is as follows:

```
map(function_to_apply, list_of_inputs)
```

You can pass the list elements to a function one by one and then collect the output. Take a look at the following example:

```
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

```
[2, 4, 6, 8]
```

Filter

The filter() method filters the given sequence through a function that tests each element in the sequence to be true or not.

```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

```
[1, 3, 5, 13]
[0, 2, 8]
```

upGrad

Reduce

The reduce() function accepts a function and a sequence, and returns a single value calculated through the following process:

1. Initially, the function is called with the first two items from the sequence, and the result is returned.
2. The function is then called again with the result obtained in step 1 and the next value in the sequence. This process keeps repeating until there are items in the sequence.

```
from functools import reduce

def do_sum(x1, x2):
    return x1 + x2

print(reduce(do_sum, [1, 2, 3, 4]))
```

```
10
```

Functions

A function is a block of organised, reusable code used to perform a single, related action. Functions provide better modularity for the application and a high degree of code reusing.
Users can create their own functions called user-defined functions.

- Defining and Calling a Function

```
def newFunction():
    print("This is Python function call")

newFunction ()
```

```
This is Python function call
```

Calling the function with an incorrect number of arguments will cause Python to throw an error.

```
def printme( str ):
  "This prints a passed string into this function"
  print str
  return;

# Now you can call printme function
printme()
```

When the previous code is executed, it produces the following result:

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Default arguments can be set in a function as follows: If default values are provided for the argument, the number of arguments can vary. Take a look at the following example.

```
def newFunction(name, age = 35):
    print("name" ,name)
    print("Age" ,age)
newFunction("First",50)
newFunction("Second")
```

```
name First
Age 50
name Second
Age 35
```

Variable length arguments: If the number of arguments is not fixed, then you can use the variable argument syntax in the function definition. If Asterix (*) is used, it is interpreted as any number of arguments.

```
def newFunction(first, *varValue):
    print("“Arguments are  :“")
    for x in varValue:
        print(x)
newFunction(1)
newFunction(1,2,3,4,5)
```

```
“Arguments are  :“
“Arguments are  :“
2
3
4
```

Lambda

A Lambda function is a small anonymous function, which can take any number of arguments. However, it can only have one expression.

```
x = lambda a : a + 10
print(x(5))
```

```
15
```

Lambda can call user-defined functions on the given input.

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

```
22
```

upGrad

Classes

A class is like an object constructor or a 'blueprint' for creating objects. Class has defined properties and functions. All classes have a function called __init__(), which is always executed when the class is being initiated. The self parameter is a reference to the current instance of the class and is used to access variables which belong to the class.

```
class FirstClass:
    """A first example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

In the last example you saw that 'i' is the property of the class and that f is the function defined in the class.

Object of Class

Accessing Object Variables

In the following example, firstObj contains the object of the class FirstClass.

Following is an example of how to access the variable inside the newly created object 'firstObj'

```
# Object of the class
firstObj =  FirstClass()
# accessing
print(firstObj.i)
print(firstObj.f())
```

```
12345
hello world
```

init and self parameter

The self parameter is a reference to the current instance of the class and is used to access variables which belong to the class.

```
##_init_ and self parameter
##The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the
class SecondClass:
    #"""A second example class"""

    def __init__(self, i):
     self.i =  i

    def f(self):
        return 'New Value of i is : ',self.i
secondObj =  SecondClass(911)
print(secondObj.f())
```

```
('New Value of i is : ', 911)
```

upGrad

Inheritance

Inheritance enables users to define a class that takes all the functionality from the parent class, and allows us to add more functionalities. Here, you will learn about how to use Inheritance in Python.

In the following example, we have a parent and a child class. The child class inherits the features of the parent class.

```
class Car:
  def __init__(self, brand, color):
    self. brand = brand
    self. color = color

  def printname(self):
    print(self.brand, self.color)

#Use the Car class to create an object, and then execute the printname method:

x = Car("Maruti", "Red")
x.printname()
```

```
Maruti Red
```

Now, let's declare the child class.

```
# Declaring the child class
class MiniVan(Car):
  pass # We are not adding any new properties and method to the class.

x = MiniVan ("Toyota", "Black")
x.printname()
```

```
Toyota Black
```

Method Overriding in Python

In Python, method overriding occurs by simply defining in the child class a method which has the same name as a method in the parent class. When you define a method in the object, you can make the latter satisfy that method call. So, the implementations of its ancestors do not come into play.

```python
# Python program to demonstrate
# method overriding


# Defining parent class
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)


# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

```
Inside Parent
Inside Child
```