Python Programming

July 12, 2017

1 1.0 Python Basics

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum.

Python comes builtin in almost all the Operating Systes, except windows. To install in windows, follow this guided installation video.

Python commands can be executed using, either: 1. Interactive Mode 2. Script Mode

Individual commands can be executed in executed in interactive mode. Script mode is preferred for write a program.

In Interactive mode, >>> indicates the prompt of the python interpreter. This python prompt, >>>, is also called as *Chevron*.

Programming in Python: 1. Interactive Mode Programming

```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Int
Type "copyright", "credits" or "license()" for more information.
>>>
```

2. Script Mode Programming

```
$ python script.py

 #!/usr/bin/python
 print "Hello, World!"

 $ chmod +x script.py
$ ./script.py
```

Python supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles.

1.1 1.1 Basic Syntax and importance of Indendation

```
In [1]: dozen = 12

In [2]: print dozen

12
```

```
In [3]: type(dozen)

Out[3]: int
```

type(object) results in the data type of the object.
Python features a dynamic type system and automatic memory management.

```
In [4]: dozen = 'twelve'

In [5]: print dozen

twelve


In [6]: type(dozen)

Out[6]: str

In [7]: print 'dozen = ', dozen

dozen =  twelve


In [8]: dozen

Out[8]: 'twelve'
```

In interactive mode, the content of the objects can be retrieved without using *print*; whereas in sript mode, *print* is essential.

```
>>>   dozen = '12'

  File "<pyshell#4>", line 2
    dozen = '12'
    ^
IndentationError: unexpected indent
>>> dozen = '12'
>>> print dozen
12
>>>

In [9]: for i in [1,2,335]:
        print i


          File "<ipython-input-9-4fc7dc342d36>", line 2
        print i
            ^
    IndentationError: expected an indented block
```

```
In [10]: for i in [1,2,335]:
             print i

1
2
335


In [11]: if 2<3:
             print "Something"
         else:
         print "Nothing"


           File "<ipython-input-11-2d945e7f1c81>", line 4
         print "Nothing"
             ^
    IndentationError: expected an indented block



In [12]: if 2<3:
             print "Something"
         else:
             print "Nothing"

Something
```

So, ensure that indentation is provided whenever it is needed, and avoid undesired indendations. Python Program works based on indentation.

PEP 8 is a python group for coding style. - They recommends **4 spaces** as indentation. - Also, they recommend to prefer spaces, to tabs. If any one is interested in using tabs, ensure that the tab space is configured to 4 spaces, in settings of your editor or IDE. - Also, there should be consistency of intendation, throughtout the program.

1.2 1.2 Reserved Keywords

```
Reserved Keywords  (27 in python 2.x)
--------------------------------------
and        assert         break          class         continue           def
elif       else           except         exec          finally            for
global     if             import         in            is                 lambda
or         pass           print          raise         return             try
yield

Reserved Keywords (33 in python 3.x)
-------------------------------------
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
```

NOTE: These reserved keywords should not be used for the names of user-defined identifiers.

1.3 1.3 Identifier Naming Conventions

Identifier can represent an object, including variables, classes, functions, execption, . . .

For Identifiers, - first character must be an alphabet (A to Z, a to z) or underscore (*).* - *from second character onwards, it can be alpha-numeric (A to Z, a to z, 0 to 9) and underscore (*) character.

```
Ex: animal, _animal, animal123, ani123mal, ani_mal123, ani12ma_13 are possible.

Ex: 123animal, animal&, \$animal, ani\$mal, 0animal are not possible. (All these re
```

- And, comma(,), dot(.), % operators are defined in python.

Naming Conventions: - Class names start with an uppercase letter. All other identifiers start with a lowercase letter. - PRIVATE identifiers start with single underscore.

```
Ex: _identierName
```

- STRONGLY PRIVATE identifiers start with two leading underscores.

  Ex: __identifierName

- Language defined Special Names - identifier with starts and ends with two underscores.

  Ex: *_init_, _main_, _file_*

Identifier casing is of two-types:

```
1. snake casing
    Ex: cost_of_mangos, result_of_router_1
2. Camel casing
    Ex: costOfMangos, resultOfRouter1
```

PEP 8 recommends to follow any one of them, but, only one type of them in a project.

NOTE: In all the following exercises and examples, Camel casing will be followed.

1.4 1.4 Case-Sensitivity

Python is **case-sensitive** language. This case-sensitivity can be removed using advanced settings, but it is strongly not recommended.

```
In [13]: animal = 'cat'
         Animal = 'cow'
```

```
In [14]: print animal

cat


In [15]: print ANIMAL


         ---------------------------------------------------------------------------
         NameError                                 Traceback (most recent call last)

         <ipython-input-15-4a6ada31f562> in <module>()
    ----> 1 print ANIMAL


         NameError: name 'ANIMAL' is not defined


In [16]: print Animal

cow
```

1.5 1.5 Quotes and Docstrings

- single ('apple' , "mango"), and triple quotes ('"apple"', """mango""")
- Triple quotes are generally used for docstrings
- Double quotes are NOT allowed. Don't be confused.
- quotes are used in defining strings
  - words, sentences, paragraphs

```
In [17]: fruit = 'apple'

In [18]: type(fruit)

Out[18]: str

In [19]: fruit = "apple"
         type(fruit)

Out[19]: str

In [20]: fruit = '''apple'''
         type(fruit)

Out[20]: str

In [21]: fruit = """apple"""
         type(fruit)
```

```
Out[21]: str
```

Inteference: Irrespecctive of the quotes, python treats all of them as 'string' data.
DocStrings

```
''' '''
""" """

In [22]: data = '''
             These are not multi-line comments, but
             are called docstrings.
             docstrinsg will be processed by the interpreter.
             triple double quotes will also work as docstrings.
         '''

In [23]: data

Out[23]: '\n    These are not multi-line comments, but\n    are called docstrings.\

In [24]: print data


    These are not multi-line comments, but
    are called docstrings.
    docstrinsg will be processed by the interpreter.
    triple double quotes will also work as docstrings.
```

1.6 1.6 Multi-Line Statements

- Line continuation operator.
- In python 2.4 or lesser versions, used as reverse division operator

```
In [25]: SomeOperation = 12+34- 1342342 + 23454545 + 3123 + \
                3455 - 3454 - 3454 - \
                234

In [26]: print "SomeOperation = ", SomeOperation

SomeOperation =  22111685
```

NOTE: Statements used within [], {}, or () doesn't need Line continuation operator

```
In [27]: SomeOperation = (12+34- 1342342 + 23454545 + 3123 +
                3455 - 3454 - 3454 -
                234)

In [28]: print "SomeOperation = ", SomeOperation

SomeOperation =  22111685
```

1.7 1.7 Mutiple Statements in a line

- ; operator is used to separate statements
- ; should not be placed after every statement, unless another statement is present in the same line.

```
In [30]: a = 12

In [31]: b = 23

In [32]: c = a + b

In [33]: print 'result = ', c

result =  35
```

All the above FOUR statements can be executed in the same line.

```
In [34]: a = 12; b = 23; c = a + b; print 'result = ', c

result =  35
```

; (semi-colon) has its importance in command line executions also.

```
C:\>python -c "print 'hello World'; a = 12; print 'a=', a"
hello World
a= 12

C:\>
```

1.8 1.8 Built-in Functions(64)

abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

```
In [35]: decade = 10

In [36]: print type(decade)  # type() returns the type of the object.

<type 'int'>

In [37]: print type(type)

<type 'type'>

In [38]: print id(decade)  # returns the address where object 'decade' is stored

4760268

In [39]: print(decade)   # print() function is different from print statement

10

In [41]: print dir(decade)   # returns the attributes and methods associated with t

['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr_

In [42]: help(decade)   # returns information and usage about the specified object,

Help on int object:

class int(object)
 |  int(x=0) -> int or long
 |  int(x, base=10) -> int or long
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is floating point, the conversion truncates towards zero.
 |  If x is outside the integer range, the function returns a long instead.
 |
 |  If x is not a number or if base is given, then x must be a string or
 |  Unicode object representing an integer literal in the given base.  The
 |  literal can be preceded by '+' or '-' and be surrounded by whitespace.
 |  The base defaults to 10.  Valid bases are 0 and 2-36.  Base 0 means to
 |  interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(...)
```

```
|       x.__abs__() <==> abs(x)
|
|   __add__(...)
|       x.__add__(y) <==> x+y
|
|   __and__(...)
|       x.__and__(y) <==> x&y
|
|   __cmp__(...)
|       x.__cmp__(y) <==> cmp(x,y)
|
|   __coerce__(...)
|       x.__coerce__(y) <==> coerce(x, y)
|
|   __div__(...)
|       x.__div__(y) <==> x/y
|
|   __divmod__(...)
|       x.__divmod__(y) <==> divmod(x, y)
|
|   __float__(...)
|       x.__float__() <==> float(x)
|
|   __floordiv__(...)
|       x.__floordiv__(y) <==> x//y
|
|   __format__(...)
|
|   __getattribute__(...)
|       x.__getattribute__('name') <==> x.name
|
|   __getnewargs__(...)
|
|   __hash__(...)
|       x.__hash__() <==> hash(x)
|
|   __hex__(...)
|       x.__hex__() <==> hex(x)
|
|   __index__(...)
|       x[y:z] <==> x[y.__index__():z.__index__()]
|
|   __int__(...)
|       x.__int__() <==> int(x)
|
|   __invert__(...)
|       x.__invert__() <==> ~x
|
```

```
|  __long__(...)
|      x.__long__() <==> long(x)
|
|  __lshift__(...)
|      x.__lshift__(y) <==> x<<y
|
|  __mod__(...)
|      x.__mod__(y) <==> x%y
|
|  __mul__(...)
|      x.__mul__(y) <==> x*y
|
|  __neg__(...)
|      x.__neg__() <==> -x
|
|  __nonzero__(...)
|      x.__nonzero__() <==> x != 0
|
|  __oct__(...)
|      x.__oct__() <==> oct(x)
|
|  __or__(...)
|      x.__or__(y) <==> x|y
|
|  __pos__(...)
|      x.__pos__() <==> +x
|
|  __pow__(...)
|      x.__pow__(y[, z]) <==> pow(x, y[, z])
|
|  __radd__(...)
|      x.__radd__(y) <==> y+x
|
|  __rand__(...)
|      x.__rand__(y) <==> y&x
|
|  __rdiv__(...)
|      x.__rdiv__(y) <==> y/x
|
|  __rdivmod__(...)
|      x.__rdivmod__(y) <==> divmod(y, x)
|
|  __repr__(...)
|      x.__repr__() <==> repr(x)
|
|  __rfloordiv__(...)
|      x.__rfloordiv__(y) <==> y//x
|
```

```
|  __rlshift__(...)
|      x.__rlshift__(y) <==> y<<x
|
|  __rmod__(...)
|      x.__rmod__(y) <==> y%x
|
|  __rmul__(...)
|      x.__rmul__(y) <==> y*x
|
|  __ror__(...)
|      x.__ror__(y) <==> y|x
|
|  __rpow__(...)
|      y.__rpow__(x[, z]) <==> pow(x, y[, z])
|
|  __rrshift__(...)
|      x.__rrshift__(y) <==> y>>x
|
|  __rshift__(...)
|      x.__rshift__(y) <==> x>>y
|
|  __rsub__(...)
|      x.__rsub__(y) <==> y-x
|
|  __rtruediv__(...)
|      x.__rtruediv__(y) <==> y/x
|
|  __rxor__(...)
|      x.__rxor__(y) <==> y^x
|
|  __str__(...)
|      x.__str__() <==> str(x)
|
|  __sub__(...)
|      x.__sub__(y) <==> x-y
|
|  __truediv__(...)
|      x.__truediv__(y) <==> x/y
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  __xor__(...)
|      x.__xor__(y) <==> x^y
|
|  bit_length(...)
|      int.bit_length() -> int
|
```

```
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

2 2.0 Arithmetic Operations

Arithmetic Operators: `+ - * / \ % ** // =`

NOTE: PEP 8 recommends to place one space around the operator

```
In [43]: var1 = 100                # int
         var2 = 2345               # int
```

2.1 2.1 Addition

```
In [44]: var3 = var1 + var2
         print var3
```

```
2445
```

```
In [50]: type(var3)                    # int + int = int

Out[50]: int

In [51]: var4 = 453453454534545435345435345345345454357090970970707    # long

In [52]: var5 = var1 + var4
         print "var5 = ", var5

var5 =  453453454534545435345435345345345454357090970970807


In [53]: var5

Out[53]: 453453454534545435345435345345345454357090970970807L
```

NOTE: Observe 'L' in the end, when the object is displayed without the print statement.

```
In [54]: type(var5)                    # int + long = long

Out[54]: long
```

Question 1: what is the largest number that can be computed in python?

```
In [55]: var6 = 10.24654565768768768798798798909356546565675767887909976540    # flo

In [56]: var6

Out[56]: 10.246545657687689

In [57]: type(var6)

Out[57]: float

In [58]: var7 = var1 + var6
         print "var7 = ", var7

var7 =  110.246545658


In [59]: type(var7)        # int + float  = float

Out[59]: float
```

Type Conversion

```
int  + int   = int
int  + long  = long
long + float = float
int  + float = float

int < long < float
```

These type conversion rules applies for other arithmetic operations too.

2.2 2.2 Subtraction

```
In [62]: 12 - 24                                        # int - int = int

Out[62]: -12

In [63]: 12 - 342344353453453456465463                  # int - long = long

Out[63]: -342344353453453456465451L

In [66]: 12.067 - 123353453443534242234453454354 3     # float - long = float

Out[66]: -1.2335345344353424e+30
```

Question 2: What is the smallest integer that can be processed by python ?

```
In [67]: 12 - 2121.2                                    # int - float = float

Out[67]: -2109.2
```

2.3 2.3 Multiplication

```
In [68]: 12 * 10                                        # int * int = int

Out[68]: 120

In [69]: 12 * 565465765765768758768696986986986986     # int * long = long

Out[69]: 6785589189189225105224363843843843832L

In [70]: 10.0 * 876876868698798797987987987987987878   # float * long = float

Out[70]: 8.76876868698799e+34

In [71]: 12 * 1.0                                       # int * float = float

Out[71]: 12.0
```

2.4 2.4 Division

There are two types of division operations in python.

```
/   division operator
//  floor division operator
\   reverse division (deprecated). It is no more used.
```

NOTE: Division is different in python 2.x and python 3.x.

```
In [72]: 10 / 2                                 # int / int = int

Out[72]: 5
```

```
In [73]: 108768768768768768 / 2          # long / int = int

Out[73]: 54384384384384384L

In [74]: 123232314423233434 / 1.0        # long / float = float

Out[74]: 1.2323231442323344e+17

In [75]: 120 / 20.0                      # int / float = float

Out[75]: 6.0
```

Question 3: what is the result of 10/3 ?

```
In [76]: 10/3                           # int /int = int

Out[76]: 3
```

Based on type-conversion rules, if both numerator and denomiator are integers, the result **SHOULD** be integer type object only.

```
In [77]: 10/3.0                         # int/float = float

Out[77]: 3.3333333333333335
```

So, in python 2.x, if the true result is essential, convert atleast one of them to float.
And, in pyth0n 3.x, int/int can also result in float-point object, based on result.

```
In [78]: float(10)    # float() is built-in function to convert any object to float-

Out[78]: 10.0

In [79]: float(10)/3                  # float/int = float

Out[79]: 3.3333333333333335

In [80]: 10/float(3)

Out[80]: 3.3333333333333335
```

Question 4: what is the difference between 10/float(3) and float(10/3)?

```
In [81]: 10/3

Out[81]: 3

In [82]: float(10/3)    # Here, the end result is converted to float

Out[82]: 3.0
```

One more example

```
In [83]: 2/10

Out[83]: 0

In [84]: 2.0/10

Out[84]: 0.2
```

reverse division operator got deprecated, and it is used as line-continuation operator now.

```
In [85]: 2\10


          File "<ipython-input-85-bb9c43a77ee4>", line 1
        2\10
            ^
    SyntaxError: unexpected character after line continuation character
```

// floor division operator

```
In [86]: 10/3

Out[86]: 3

In [87]: 10//3

Out[87]: 3

In [88]: 10/3.0

Out[88]: 3.3333333333333335

In [89]: 10//3.0        # 3.3 is between integers 3 and 4

Out[89]: 3.0
```

NOTE: floor division results in the floor value of the division result

```
In [90]: import math
         math.floor(10/3)

Out[90]: 3.0

In [91]: -10/3.0

Out[91]: -3.3333333333333335

In [92]: -10//3.0       #  -4 < -3.3333 < -3

Out[92]: -4.0
```

Another example

```
In [93]: 10/4.0

Out[93]: 2.5

In [94]: 10//4.0       # 2 < 1.5 < 3

Out[94]: 2.0

In [95]: 10/-4.0

Out[95]: -2.5

In [96]: 10//-4.0     # -3 < -2.5 < -2

Out[96]: -3.0
```

2.5 2.5 Power Operation

** is the power operator.

pow() built-in function for power operation

```
In [97]: 2 ** 3                                                    # int ** int = int

Out[97]: 8

In [98]: 22222222222222222 ** 3                                    # long ** int = long

Out[98]: 10973936899862825459533607681755833196159122085048L

In [99]: 22222222222222222 ** 3.0                                  # long ** float = floa

Out[99]: 1.0973936899862828e+49

In [100]: 2 ** 3.0                                                  # int ** float = floa

Out[100]: 8.0

In [101]: pow(2,3)

Out[101]: 8

In [102]: pow(4,0.5)   # square-root operation

Out[102]: 2.0

In [103]: pow(4, 76876878678768768)
```

```
---------------------------------------------------------------------------

MemoryError                               Traceback (most recent call last)

<ipython-input-103-52707f8062f9> in <module>()
----> 1 pow(4, 7687687867876876878)


MemoryError:
```

NOTE: It may lead to Memory Error or Overflow Error, if the result exceeds its capacity.

2.6 2.6 Exponent Operation

```
In [104]: 1e1    # equal to 1.0 * 10 **1

Out[104]: 10.0

In [105]: 1.0 * 10 **1

Out[105]: 10.0

In [106]: 1 * 10 **1

Out[106]: 10

In [107]: 2e3      # 2.0 * 10 ** 3

Out[107]: 2000.0

In [109]: -4.6e3

Out[109]: -4600.0
```

2.7 2.7 Modulo Operation

- It results in the remainder after division

```
In [110]: 0%3, 1%3, 2%3, 3%3, 4%3, 5%3, 6%3, 7%3, 8%3, 9%3, 10%3

Out[110]: (0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

Observe that there are 3 elements repeating (0, N-1) where N is modulo divisor.
You can take the analogy of an Analog clock. After it completes 12 hours, it starts again from 0

```
In [111]: 0%12, 1%12, 11%12

Out[111]: (0, 1, 11)
```

```
In [112]: 12%12

Out[112]: 0

In [113]: 14%12  # 2 past noon

Out[113]: 2

In [114]: 16.45%12           # Observe the output precision; it is 16 digits post d

Out[114]: 4.449999999999999

In [115]: -18%12

Out[115]: 6

In [116]: -18%-12

Out[116]: -6

In [117]: 18%-12

Out[117]: -6
```

Inference: sign of modulo number is reflected.

2.8 2.8 Complex Numbers

Complex Number = Real Number +/- Imaginary Number

In python, **'j'** is used to represent the imaginary number.

```
In [119]: n1 = 2 + 3j

In [120]: print n1

(2+3j)


In [121]: print type(n1)

<type 'complex'>


In [123]: n2 = 3 - 2j

          print "n2 = ", n2

          print "type(n2) = ", type(n2)

n2 =  (3-2j)
type(n2) =  <type 'complex'>
```

```
In [124]: print "n1       =", n1
          print "n1 + n2 = ", n1 + n2

n1       = (2+3j)
n1 + n2 =  (5+1j)


In [125]: print "n1 - n2 = ", n1 - n2

n1 - n2 =  (-1+5j)


In [126]: n1/n1

Out[126]: (1+0j)

In [127]: pow(n1,2)

Out[127]: (-5+12j)

In [128]: print n1, n1.conjugate()    # Observe the signs of imaginary numbers

(2+3j) (2-3j)


In [129]: print n1, n1.real, n1.imag

(2+3j) 2.0 3.0


In [130]: n2 = 2.0 - 3.45j

In [131]: n2.real

Out[131]: 2.0

In [132]: n2.imag

Out[132]: -3.45

In [133]: 4j

Out[133]: 4j
```

NOTE: 4*j, j4, j*4 are not possible. In these cases, interpreter treats 'j' as a variable.

```
In [135]: print n1*n2.real, (n1*n2).real

(4+6j) 14.35


In [136]: n1.real+n2.imag
```

```
Out[136]: -1.4500000000000002

In [137]: n1.real+n2.imag * j


        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)

        <ipython-input-137-d45ccd14acf2> in <module>()
    ----> 1 n1.real+n2.imag * j


        NameError: name 'j' is not defined


In [138]: n1.real+n2.imag * 1j

Out[138]: (2-3.45j)

In [139]: complex(2,-3.456)       # complex()  - Builtin function

Out[139]: (2-3.456j)

In [140]: (3 + 4j) == (4j + 3)    # == checks value equivalence

Out[140]: True

In [141]: (3 + 4j) is (4j + 3)    # is - checks object level (both value and addres

Out[141]: False
```

2.9 2.9 abs()

Builtin function, to return the absolute value.
If a is positive real integer,

```
abs(a)  = a
abs(-a) = a
abs((a+bj))  is equal to math.sqrt(pow(a,2), pow(b,2))

In [142]: abs(3)

Out[142]: 3

In [143]: abs(-3)

Out[143]: 3

In [144]: abs(3+4j)
```

```
Out[144]: 5.0

In [145]: import math

          print math.sqrt(3**2+4**2)

5.0
```

2.10 2.10 devmod()

divmod(x,y) returns x//y, x%y

```
In [147]: divmod(10,2)

Out[147]: (5, 0)

In [148]: divmod(10,3)

Out[148]: (3, 1)

In [149]: divmod( 3+4j, 2)

c:\python27\lib\site-packages\ipykernel\__main__.py:1: DeprecationWarning: complex
  if __name__ == '__main__':


Out[149]: ((1+0j), (1+4j))
```

NOTE: Observe the warning. In python, warnings are disabled, by default.

2.11 2.11 Compound(mixed) Operations

+=, -=, *=, /=, **=, %=, <<=, >>=, ^=, |=

NOTE: Pre- and Post- increment/ decrements (++a, a++, –a, a–) are not valid in Python

```
In [150]: a = 12
          print a, type(a)

12 <type 'int'>


In [151]: a = a + 1 ; print "a = ", a    # ; is used to write multiple statement in

a =  13


In [152]: a += 1 ; print "a = ", a

a =  14
```

```
In [153]: a -= 1 ; print "a = ", a    # a = a -1

a =  13

In [154]: a *= 2 ; print "a = ", a   # a = a*2

a =  26

In [155]: a /= 2 ; print "a = ", a   # a = a / 2

a =  13

In [156]: a **= 2 ; print "a = ", a   # a = a ** 2

a =  169

In [157]: a %= 100 ; print "a = ", a

a =  69

In [158]: a = 23; a//=2; print "a = ", a

a =  11

In [159]: a <<= 1; print "a = ", a    # left-shift

a =  22

at binary level 128 64 32 16 8 4 2 1
                           1 0 1 1
            <<1    0  0  0  1 0 1 1 0

In [160]: a >>= 1; print "a = ", a    # right-shift

a =  11

at binary level 128 64 32 16 8 4 2 1
             11    0  0  0  0 1 0 1 1
              2    0  0  0  0 0 0 1 0
        ----------------------------
              ^    0  0  0  0 1 0 0 1    9

In [161]: a^=2; print "a = ", a    # bitwise XOR operation
```

```
a =  9


at binary level 128 64 32 16 8 4 2 1
               9   0  0  0  0 1 0 0 1
               2   0  0  0  0 0 0 1 0
         ----------------------------
               |   0  0  0  0 1 0 1 1    11
```

```
In [162]: a|=2; print "a = ", a    # bitwise OR operation

a =  11
```

2.12 2.12 Working in Script Mode

```
In [163]: #!/usr/bin/python
          # This is called shebong line

          # prog1.py

          print "Hello World!"

Hello World!
```

```
In [164]: #!/usr/bin/python

          '''
              DocStrings must come immediately after shebang line.
              These are not multi-line comments, but
              are called docstrings.
              docstrinsg will be processed by the interpreter.
              triple double quotes will also work as docstrings.
          '''

          # prog2.py

          # This hash/pound is the comment operator, used for
          # both single line and multi-line comments.
          # comment line will be ignored by interpreter

          #either single, single or double quotes, can be used for strings


          costOfMango = 12
          print "cost Of Each Mango is ", costOfMango
          costOfApple = 40
          print "cost Of Each Apple is ", costOfApple
```

```
            # what is the cost of dozen apples and two dozens of mangos

            TotalCost = 12* costOfApple + 2*12* costOfMango

            print "Total cost is ", TotalCost
```

```
cost Of Each Mango is  12
cost Of Each Apple is  40
Total cost is  768
```

```
In [165]: #!/usr/bin/python

            """
            Purpose: Demo

            """
            # prog3.py

            # print is a statement in python 2, and is a function call in python 3

            # now, python 2 is supporting both

            print "Hello World!"
            print("Hello World!")


            # by default, print will lead to display in next line

            print "This is",    # , after print will suppress the next line
                                 # but, a space will result
            print "python class"

            # PEP 8 recommends to use only print statement or function call throughto

            # ; semicolon operator
            # It is used as a statement separator.

            name = 'yash'
            print 'My name is ', name

            name = 'yash'; print 'My name is ', name

            print "who's name is ", name, '?'
```

```
Hello World!
Hello World!
```

```
This is python class
My name is  yash
My name is  yash
who's name is  yash ?
```

```
In [166]: #!/usr/bin/python

          """
          Purpose: Handling Quotes

          """
          # prog4.py

          print "'"
          print '"'
          print '\''

          print "''' '''"

          print '"""'
          print "'''"

          print ''' """ """ '''
          print """ ''' ''' """
```

```
'
"
'
''' '''
""
''
 """ """
 ''' '''
```

2.13 2.13 Operator precedence in python

It follows **PEMDAS** rule, and left to right, and top to bottom.

```
P - Paranthesis
E - Exponent
M - Multiplication
D - Division
A - Addition
S - Subtraction
```

Every type of braces has importance in python.

```
 {}   - used for dictionaries and sets
 []   - used for lists
 ()   - used of tuples; also used in arithmetic operations
```

```
In [167]: #!/usr/bin/python
          """
          Purpose: Demonstration of operator precedence
          """
          # prog5.py

          result1 = (22+ 2/2*4//4-89)
          print "result1 = ", result1

          result2 = 32/2/2/2/2
          print "result2 = ", result2

          result3 = 2 + (3.0 - 5j).imag + 2 ** 3 * 2
          print "result3 = ", result3
```

```
result1 =  -66
result2 =  2
result3 =  13.0
```

Assignment 1: Examine the operator precedence with other expressions

2.14 2.14 IO Operations

In python 2.x, raw_input() and input() are two builtin functions used for getting runtime input.

```
raw_input()  - takes any type of runtime input as a string.

input()      - takes any type of runtime input originally without any type conversi
```

NOTE: Working with raw_input() requires us to use type converters to convert the data into the required data type.

In Python 3.x, there is only input() function; but not raw_input(). The Job of raw_input() in python 2.x is done by input() in python 3.x.

```
In [168]: #!/usr/bin/python

          """
              Purpose : demonstration of input() and raw_input()

          """
          # prog6.py

          dataRI = raw_input('Enter Something: ')
```

```
        dataI = input('Enter something: ')

        print "dataRI = ", dataRI, " type(dataRI) = ", type(dataRI)

        print "dataI = ", dataI, " type(dataI) = ", type(dataI)
```

```
Enter Something: 999
Enter something: 999
dataRI =  999  type(dataRI) =  <type 'str'>
dataI =  999  type(dataI) =  <type 'int'>
```

Analyzed outputs for various demonstrated cases:

```
>>>
==================== RESTART: C:/pyExercises/class3_io.py ====================
Enter Something: 123
Enter something: 123
123 <type 'str'>
123 <type 'int'>
>>>
==================== RESTART: C:/pyExercises/class3_io.py ====================
Enter Something: 'Yash'
Enter something: 'Yash'
'Yash' <type 'str'>
Yash <type 'str'>
>>>
==================== RESTART: C:/pyExercises/class3_io.py ====================
Enter Something: True
Enter something: True
True <type 'str'>
True <type 'bool'>
>>>
==================== RESTART: C:/pyExercises/class3_io.py ====================
Enter Something: Yash
Enter something: Yash

Traceback (most recent call last):
  File "C:/pyExercises/class3_io.py", line 12, in <module>
    dataI = input('Enter something: ')
  File "<string>", line 1, in <module>
NameError: name 'Yash' is not defined
>>> dataRI
'Yash'
```

input() takes only qualified data as runtime input. Whereas **raw_input()** will qualify any data as a 'str' type.

```
In [170]: #!/usr/bin/python
```

```
        """
            Purpose : demonstration of input() and raw_input()

        """
        # prog7.py

        dataRI = int(raw_input('Enter a number: '))

        dataI = input('Enter a number: ')


        print "dataRI = ", dataRI, " type(dataRI) = ", type(dataRI)

        print "dataI = ", dataI, " type(dataI) = ", type(dataI)

        print "Sum of numbers is ", dataRI+dataI
```

```
Enter a number: 999
Enter a number: 999
dataRI =  999  type(dataRI) =  <type 'int'>
dataI =  999  type(dataI) =  <type 'int'>
Sum of numbers is  1998
```

Analyzed outputs for various demonstrated cases:

```
================== RESTART: C:/pyExercises/class3_io1.py ==================
Enter a number: 123
Enter a number: 123
123 <type 'str'>
123 <type 'int'>
>>>
================== RESTART: C:/pyExercises/class3_io1.py ==================
Enter a number: 123
Enter a number: 123
123 <type 'str'>
123 <type 'int'>
Sum of numbers is

Traceback (most recent call last):
  File "C:/pyExercises/class3_io1.py", line 19, in <module>
    print "Sum of numbers is ", dateRI+dataI
NameError: name 'dateRI' is not defined
>>>
================== RESTART: C:/pyExercises/class3_io1.py ==================
Enter a number: 123
Enter a number: 123
123 <type 'str'>
```

```
123 <type 'int'>
Sum of numbers is

Traceback (most recent call last):
  File "C:/pyExercises/class3_io1.py", line 19, in <module>
    print "Sum of numbers is ", dataRI+dataI
TypeError: cannot concatenate 'str' and 'int' objects
>>>
================== RESTART: C:/pyExercises/class3_io1.py ==================
Enter a number: 123
Enter a number: 123
123 <type 'int'>
123 <type 'int'>
Sum of numbers is  246
>>>
```

Inference: - input() takes only qualified objects as inputs; whereas raw_input() considers any input as string data. - input() processess the data before taking as input; It is sensed as a security threat by many developers.

3 3.0 String Operations

String data type can be representing using either single or double quotes.

3.1 3.1 Creating string(s)

```
In [172]: s1 = 'Python Programming'      # single quotes

In [173]: s1

Out[173]: 'Python Programming'

In [174]: print s1

Python Programming


In [175]: print type(s1)

<type 'str'>


In [176]: s2 = "Django"                  # double quotes

In [177]: print s2, type(s2)

Django <type 'str'>


In [178]: s3 = ''' python programming with Django ''' # triple single quotes
```

```
In [179]: print s3

 python programming with Django


In [180]: print type(s3)

<type 'str'>


In [181]: s4 = """ python programming with Django  """  # triple double quotes

In [182]: print s4

 python programming with Django


In [183]: type(s4)

Out[183]: str

In [184]: print django1

        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-184-5701398af8ee> in <module>()
    ----> 1 print django1


        NameError: name 'django1' is not defined


In [185]: print 'django1'

django1


In [186]: s5 = '~!@#$%^& *()1232425'  # Any special character can be taken within s

In [187]: print s5, type(s5)

~!@#$%^& *()1232425 <type 'str'>


In [188]: s6 = str(123.34)            # str() is a builtin function to convert to s

In [189]: print s6, type(s6)
```

```
123.34 <type 'str'>
```

```
In [190]: s7 = str(True)
```

```
In [191]: print s7, type(s7)
```

```
True <type 'str'>
```

3.2 3.2 Indexing

```
In [192]: print s1
```

```
Python Programming
```

```
In [193]: print len(s1)    # len() is a bultin function to return the length of obje
```

```
18
```

```
P y t h o n       P   r   o   g   r   a   m   m   i   n   g
0 1 2 3 4 5   6   7   8   9 10 11 12 13 14 15 16 17  -> forward indexing
            -12  -11  -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  -> Reverse indexing
```

```
In [194]: s1[0]
```

```
Out[194]: 'P'
```

```
In [195]: s1[6]    # white-space character
```

```
Out[195]: ' '
```

```
In [196]: s1[17]
```

```
Out[196]: 'g'
```

```
In [197]: s1[18]
```

```
        ---------------------------------------------------------------------------
        IndexError                                Traceback (most recent call last)

        <ipython-input-197-38bfda003eb6> in <module>()
    ----> 1 s1[18]


        IndexError: string index out of range
```

NOTE: Indexing can be done from 0 through len(string)-1

```
In [198]: s1[-1]

Out[198]: 'g'

In [199]: s1[-5]

Out[199]: 'm'

In [200]: s1[-16]

Out[200]: 't'

In [201]: s1[-18] == s1[0]

Out[201]: True

In [202]: s1[-len(s1)] == s1[0]

Out[202]: True

In [204]: len(s1)/2

Out[204]: 9

In [203]: s1[len(s1)/2] == s1[-len(s1)/2]

Out[203]: True

In [205]: sample = 'battles'
          print len(sample)

7


In [206]: len(sample)/2

Out[206]: 3

In [209]: sample[len(sample)/2] == sample[-len(sample)/2]

Out[209]: True
```

Question 5: What is string[-0]?

```
In [210]: sample[-0]  # is equal to sample[0]

Out[210]: 'b'

In [211]: sample[0]

Out[211]: 'b'
```

3.3 3.3 String Slicing

```
In [212]: print s1

Python Programming


In [213]: s1[2:6]     # string[InitialBound, finalBound]

Out[213]: 'thon'

In [214]: s1[2:8]

Out[214]: 'thon P'

In [215]: s1[2:17]

Out[215]: 'thon Programmin'

In [216]: s1[2:18]

Out[216]: 'thon Programming'

In [217]: s1[2:786] # Observe the finalBound

Out[217]: 'thon Programming'

In [218]: s1[17:786]

Out[218]: 'g'

In [219]: s1[18:786]

Out[219]: ''

In [220]: s1[2:]  # default finalBound corresponds to lastCharacter in string

Out[220]: 'thon Programming'

In [221]: s1[:]   # defaul initialBound is 0th element

Out[221]: 'Python Programming'

In [222]: s1[:-1]

Out[222]: 'Python Programmin'

In [223]: s1[-5:-1]

Out[223]: 'mmin'

In [224]: s1[-5:17]    # complex indexing
```

```
Out[224]: 'mmin'

In [225]: s1[::1]

Out[225]: 'Python Programming'

In [226]: s1[::2]

Out[226]: 'Pto rgamn'

In [227]: s1[::3]

Out[227]: 'Ph oai'

In [228]: s1[::4]

Out[228]: 'Poran'

In [229]: s1[::-1]

Out[229]: 'gnimmargorP nohtyP'

In [230]: s1[::] # default step is +1

Out[230]: 'Python Programming'

In [231]: s1[:] == s1[::]

Out[231]: True

In [232]: s1[4:9]

Out[232]: 'on Pr'

In [233]: s1[4:9:1]     # string[initialBound, finalBound, increment/decrement]

Out[233]: 'on Pr'

In [234]: s1[4:9:-1]    # 4-1 = 3  index 3 is not represented in this object

Out[234]: ''
```

NOTE: After all these alterations, the original string object will not change, until it is overwrited.

3.4 3.4 Mutability of Strings

String objects are immutable. They, can't be edited. Only way is to overwrite it

```
In [235]: print s1

Python Programming


In [236]: s1[3]

Out[236]: 'h'

In [237]: s1[3] = 'H'


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-237-a32a26674f0e> in <module>()
    ----> 1 s1[3] = 'H'


        TypeError: 'str' object does not support item assignment


In [238]: id(s1)  # to get the stored address location

Out[238]: 60867072

In [239]: s1 = "PytHon Programming" # object overwriting taken place

          print s1

PytHon Programming


In [240]: id(s1)

Out[240]: 61110096
```

Inference: By observing the id(s1) in the above two results, it is concluded that overwriting an object creates new object in a different location.

3.5 3.5 String attributes

```
In [241]: print dir(s1)

['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__for
```

```
In [242]: s1 = 'PyTHON PROGRAMMING'
          print s1

PyTHON PROGRAMMING


In [243]: s1.count('m')

Out[243]: 0

In [244]: s1.count('M')

Out[244]: 2

In [245]: s1.endswith('ing')  # endswith() returns the boolen result

Out[245]: False

In [246]: s1.endswith('ING')

Out[246]: True

In [247]: s1.startswith('Py')

Out[247]: True

In [248]: s1.find('P')

Out[248]: 0

In [249]: s1.find('THON')

Out[249]: 2

In [250]: s1.find('MM')

Out[250]: 13

In [251]: s1.find('M')

Out[251]: 13

In [252]: s1.index('THON')

Out[252]: 2
```

Question: Difference between s1.find() and s1. rfind()?

```
In [253]: s1.rfind('P')

Out[253]: 7
```

```
In [254]: s1.rfind('THON')

Out[254]: 2

In [255]: s1.rfind('MM')

Out[255]: 13

In [256]: s1.rfind('M')

Out[256]: 14

In [257]: s1.index('THON')

Out[257]: 2
```

Question: Difference between s1.find() and s1. index()?

```
In [258]: s1.find('Q')

Out[258]: -1

In [259]: s1.index('Q')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-259-28e3ca7ec558> in <module>()
----> 1 s1.index('Q')

ValueError: substring not found

In [260]: s1

Out[260]: 'PyTHON PROGRAMMING'

In [261]: s1.capitalize()

Out[261]: 'Python programming'

In [262]: s1  # Observe that the original object wasn't changed

Out[262]: 'PyTHON PROGRAMMING'

In [263]: s1.lower()

Out[263]: 'python programming'
```

```
In [264]: s1.upper()

Out[264]: 'PYTHON PROGRAMMING'

In [265]: s1.title()

Out[265]: 'Python Programming'

In [266]: s1.swapcase()

Out[266]: 'pYthon programming'
```

Question: what is the data type of result of string.split() ?

```
In [267]: s1.split(' ')    # results in 'list' datatype

Out[267]: ['PyTHON', 'PROGRAMMING']

In [268]: s1.split('O')

Out[268]: ['PyTH', 'N PR', 'GRAMMING']

In [269]: s1    # Observe that the original object is unchanged

Out[269]: 'PyTHON PROGRAMMING'

In [270]: s1.split('N')         # string to list conversion

Out[270]: ['PyTHO', ' PROGRAMMI', 'G']

In [271]: s1.split('r')         # results in a list type, even if the character is

Out[271]: ['PyTHON PROGRAMMING']

In [272]: s1.split('y')

Out[272]: ['P', 'THON PROGRAMMING']

In [273]: len(s1.split('y'))

Out[273]: 2

In [274]: ''.join(s1.split('y'))    # list to strinsg conversion

Out[274]: 'PTHON PROGRAMMING'

In [275]: '@'.join(s1.split('y'))   # delimiter can be placed

Out[275]: 'P@THON PROGRAMMING'

In [276]: s1.split('O')

Out[276]: ['PyTH', 'N PR', 'GRAMMING']
```

```
In [277]: '@'.join(s1.split('O'))   # Observe that 'O' is replaced by '@'.   This i

Out[277]: 'PyTH@N PR@GRAMMING'

In [278]: s9  = '''
                  This is a good day!
                  Fall 7 times, raise 8!
                  This is a famous japanese quote.
                      '''

In [279]: print len(s9), s9

114
        This is a good day!
        Fall 7 times, raise 8!
        This is a famous japanese quote.



In [280]: print 'IS'.join(s9.split('is'))


        ThIS IS a good day!
        Fall 7 times, raISe 8!
        ThIS IS a famous japanese quote.



In [281]: print 'IS'.join(s9.split(' is'))


        ThisIS a good day!
        Fall 7 times, raise 8!
        ThisIS a famous japanese quote.



In [282]: print ' IS'.join(s9.split(' is'))


        This IS a good day!
        Fall 7 times, raise 8!
        This IS a famous japanese quote.



In [283]: s1

Out[283]: 'PyTHON PROGRAMMING'

In [ ]:
```