Content Delivered in class2_30-July-2016

- Chapter 1: An Introduction to Python
  - Working with IDLE
  - Getting Help in python
- Chapter 2: Basics
  - Basic Syntax and semantics in python
  - Importance of indentation, & ways to identify them
  - Importance and difference between the 'Interactive mode' and 'script mode'
  - Single line and multi-line commenting
  - Python Reserved Words
  - Various types of Quotes, and their importance
  - Arithmetic Operations
  - Insight to PEP 8 recommendations

Assignments given:

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

Assignment 2: what is the smallest number that can be computed in python?

Assignment 3: examine the operator precedence will other examples

---

2.0 Python Basics

IDLE will be installed, along with basic python in Windows. In Linux and Unix, it can be installed manually. IDLE is a python IDE, from Python. Python commands can be executed using, either:

```
1. Interactive Mode, or
2. Script Mode
```

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

In script mode, >>> indicates the prompt of the python interpreter.

```
Programming in Python:
    1. Interactive Mode Programming
            Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500
 32 bit (Intel)] on win32
            Type "help", "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
```

2.1 Basic Syntax and Indenting

```
In [1]: a = 12
```

```
In [2]: a
```

```
Out[2]: 12
```

```
In [3]: b = 34
```

```
>>>  a=12
  File "<stdin>", line 1
    a=12
    ^
IndentationError: unexpected indent
```

```
>>> for i in [1,2,335]:
... print i
  File "<stdin>", line 2
    print i
        ^
IndentationError: expected an indented block
```

In [6]:

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

```
1
2
335
```

In [12]:

```
if True:
        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. It 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.

2.2 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_l3 are possible

Ex: 123animal, animal&, $animal, ani$mal, 0animal are not possible. (All these resu
lt in SyntaxError)

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

    Naming Conventions
        - Class names start with an uppercase letter. All other identifiers start w
ith 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__
```

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

In [8]:
```
animal = "Cat"
```

In [9]:
```
Animal = "Cow"
```

```
In [10]: animal

Out[10]: 'Cat'
```

```
In [11]: Animal

Out[11]: 'Cow'
```

Identifier casing is of two-types:

```
1. snake casing
    ex: cost_of_mangos
2. Camel casing
    ex: costOfMangos
```

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

comment operator

```
# comment Operator
Interpretor ignore the line, right to this operator
The is only line comment, in python.
```

Docstrings

```
'''  '''

"""  """
```

```
In [23]: '''
             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.
         '''

Out[23]: '\n    These are not multi-line comments, but\n    are called docstrings.\n
           docstrinsg will be processed by the interpreter.\n    triple double quotes
          will also work as docstrings.\n'
```

Quotes

```
- 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
```

Multi-Line Statements

```
- \ Line continuation operator. (Also, used as reverse division operator)
```

In [24]:
```
sum = 12+34- 1342342 + 23454545 + 3123 + \
                       3455 - 3454 - 3454 - \
                       234
```

In [25]:
```
sum
```

Out[25]:
```
22111685
```

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

In [26]:
```
months = ('Jan', 'Feb', 'Mar',
                          'Apr', 'May', 'Jun',
                          'jul', 'Aug')
```

In [27]:
```
months
```

Out[27]:
```
('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'jul', 'Aug')
```

Mutiple Statements in a line

```
- ; operator is used to separate statements
```

In [28]:
```
a = 12; b = 34; a+b
```

Out[28]:
```
46
```

2.3 Reserved Keywords in Python:

```
Reserved Keywords  (27 in python 2.x)
--------------------------------------
and        assert         break          class         continue           def
   del
elif       else           except         exec          finally            for
   from
global     if             import         in            is                 lambda
   not
or         pass           print          raise         return             try
   while
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
```

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

```
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 [14]: a = 12
         type(a)  # type() returns the type of the object.

Out[14]: int
```

```
In [15]: type(type)

Out[15]: type
```

```
In [16]: id(a)  # returns the address where object 'a' is stored

Out[16]: 39351820
```

In [19]:

```
print(a)
```

```
12
```

In [20]:

```
print(dir(a))  # returns the attributes and methods associated with the object
 'a'
```

```
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__de
lattr__', '__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '_
_format__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__i
ndex__', '__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mo
d__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__', '_
_pos__', '__pow__', '__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__red
uce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod_
_', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub_
_', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__su
b__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_lengt
h', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

The remaining built-in functions will be dealt appropriately in their corresponding context.

In [22]:

```
help(a)   # returns information and usage about the specified object, or funct
ion, ..
```

```
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 zer
o.
 |  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
```

2.4 Arithmetic Operations

Arithmetic Operators:

```
+ - * / \ % ** // =
```

PEP 8 recommends to place one space around the operator

```
In [29]: var1 = 123
```

```
In [30]: var2 = 2345
```

Addition

```
In [31]: var1+var2

Out[31]: 2468
```

```
In [32]: var3 = 23.45
```

```
In [33]: type(var1), type(var2), type(var3)

Out[33]: (int, int, float)
```

```
In [34]: var1+var3 # type-casting takes place  # int + float = float

Out[34]: 146.45
```

```
In [35]: var4 = 45345345453454543534543534534534545435
```

```
In [36]: type(var4)

Out[36]: long
```

```
In [37]: var4

Out[37]: 45345345453454543534543534534534545435L
```

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

```
In [38]: var2+var4   # int + long int

Out[38]: 45345345453454543534543534534534547780L
```

```
In [39]: var3 + var4  # float + long int

Out[39]: 4.534534545345455e+37
```

```
In [40]: var2

Out[40]: 2345
```

```
In [41]: var2 = 234.456   # overwrite the existing object; dynamic typing
```

```
In [42]: var2

Out[42]: 234.456
```

```
In [43]: type(var2)

Out[43]: float
```

subtraction

```
In [44]: var1 - var2

Out[44]: -111.45599999999999
```

```
In [45]: var2 - var4

Out[45]: -4.534534545345455e+37
```

Assignment 2: what is the smallest number that can be computed in python?

Multiplication

```
In [46]: var1*var2  # int * int

Out[46]: 28838.088
```

```
In [47]: var1*var2  # int * float

Out[47]: 28838.088
```

```
In [48]: var1, var2, var3, var4

Out[48]: (123, 234.456, 23.45, 45345345453454543534543534534534545435L)
```

```
In [49]: var5 = 23
```

```
In [50]: var1*var5 # int * int

Out[50]: 2829
```

```
In [51]: var1 * var4 # int * long int

Out[51]: 5577477490774908854748854747747749088505L
```

Division Operation

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

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

```
In [52]: 10/5

Out[52]: 2
```

```
In [53]: 10/2

Out[53]: 5
```

```
In [54]: 10/3

Out[54]: 3
```

```
In [55]: 10//3

Out[55]: 3
```

```
In [56]: 10/3.0   # true division in python 2

Out[56]: 3.3333333333333335
```

In python3, 10/3 will give true division

```
In [57]: 2/10

Out[57]: 0
```

\ reverse division operator got deprecated

```
In [59]: 2\10

           File "<ipython-input-59-1bdd425914b1>", line 1
             2\10
                       ^
         SyntaxError: unexpected character after line continuation character
```

```
In [60]: 10/3

Out[60]: 3
```

```
In [61]: 10/3.0

Out[61]: 3.3333333333333335
```

```
In [62]: 10.0/3

Out[62]: 3.3333333333333335
```

10.0/3.0

```
In [63]: float(3)   # float() is a built-in function, used to convert to floating-point
          value

Out[63]: 3.0
```

```
In [64]: 10/float(3)

Out[64]: 3.3333333333333335
```

```
In [65]: 2/10

Out[65]: 0
```

```
In [66]: 2.0/10

Out[66]: 0.2
```

```
In [67]: 2.0//10

Out[67]: 0.0
```

```
In [68]: 5/2.0

Out[68]: 2.5
```

```
In [69]: 5//2.0

Out[69]: 2.0
```

```
In [70]: 5//2    float division will convert to floor(), after division

Out[70]: 2
```

power operation

```
In [71]: 2 ** 3

Out[71]: 8
```

```
In [72]: 3**100

Out[72]: 515377520732011331036461129765621272702107522001L
```

```
In [73]: pow(2,3)

Out[73]: 8
```

```
In [74]: pow(4,0.5)  # square root

Out[74]: 2.0
```

Power Operation can't do as var2 is float type

```
In [75]: var1**var2

         ---------------------------------------------------------------------------
         OverflowError                             Traceback (most recent call last)
         <ipython-input-75-e9b276a0d761> in <module>()
         ----> 1 var1**var2

         OverflowError: (34, 'Result too large')
```

```
In [76]: a,b = 56, 23    # tuple unpacking
```

```
In [77]: a

Out[77]: 56
```

```
In [78]: b

Out[78]: 23
```

```
In [79]: a**b

Out[79]: 16155656889615734329398214425629966729216L
```

```
In [80]: pow(a,b)

Out[80]: 16155656889615734329398214425629966729216L
```

exponent operation

```
In [81]: 1e10

Out[81]: 10000000000.0
```

```
In [82]: 1e1  # equal to 1 * 10 **1

Out[82]: 10.0
```

```
In [83]: 1 * 10 **1

Out[83]: 10
```

```
In [84]: 1.0 * 10 **1

Out[84]: 10.0
```

Working in Script Mode

```
In [87]: #!/usr/bin/python
         # This is called shebong line

         # prog1.py

         print "Hello World!"

         Hello World!
```

```
#!/usr/bin/python

# 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

'''
    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.
'''

#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


# 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 throughtout th
e project

# ; 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, '?'

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

print "'''  '''"

print '""'
print "''"

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

```
cost Of Each Mango is  12
cost Of Each Apple is  40
Total cost is  768
Hello World!
Hello World!
This is python class
My name is  yash
My name is  yash
who's name is  yash ?
'
"
'
'''  '''
""
''
 """ """
 ''' '''
```

In [89]:

```
#!/usr/bin/python

# prog3.py

# 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


result = (22+ 2/2*4//4-89)
print result
```

```
-66
```

Assignment 3: examine the operator precedence will other examples