Functions

Python is **pass by reference**, function arguments are passed by reference.

- Basic Form :

```
def func1(posArg1, keywordArg1 =
1, ..):
```

Note :
- Keyword arguments MUST follow positional arguments.
- Python by default is NOT "lazy evaluation", expressions are evaluated immediately.

- Function Call Mechanism :
  1. All functions are local to the module level scope. See 'Module' section.
  2. Internally, arguments are packed into a tuple and dict, function receives a tuple 'args' and dict 'kwargs' and internally unpack.
- Common usage of 'Functions are objects' :

```
def func1(ops = [str.strip, user_
define_func, ..], ..):
      for function in ops:
          value = function(value)
```

RETURN VALUES

- **None** is returned if end of function is reached without encountering a return statement.
- Multiple values return via ONE tuple object

```
return (value1, value2)
value1, value2 = func1(..)
```

ANONYMOUS (AKA LAMBDA) FUNCTIONS

- What is Anonymous function?
  A simple function consisting of a single statement.

```
lambda x : x * 2
# def func1(x) : return x * 2
```

- Application of lambda functions : 'curring' aka deriving new functions from existing ones by partial argument application.

```
ma60 = lambda x : pd.rolling_mean(x,
60)
```

USEFUL FUNCTIONS (FOR DATA STRUCTURES)

1. **Enumerate** returns a sequence (i, value) tuples where i is the index of current item.

```
for i, value in enumerate(collection):
```

   - Application : Create a dict mapping of value of a sequence (assumed to be unique) to their locations in the sequence.

2. **Sorted** returns a new sorted list from any sequence

```
sorted([2, 1, 3]) => [1, 2, 3]
```

   - Application :

```
sorted(set('abc bcd')) => [' ',
'a', 'b', 'c', 'd']
# returns sorted unique characters
```

3. **Zip** pairs up elements of a number of lists, tuples or other sequences to create a list of tuples :

```
zip(seq1, seq2) =>
[('seq1_1', 'seq2_1'), (..), ..]
```

   - Zip can take arbitrary number of sequences. However, the number of elements it produces is determined by the 'shortest' sequence.
   - Application : Simultaneously iterating over multiple sequences :

```
for i, (a, b) in
enumerate(zip(seq1, seq2)):
```

   - Unzip - another way to think about this is converting a list of rows to a list of columns.

```
seq1, seq2 = zip(*zipOutput)
```

4. **Reversed** iterates over the elements of a sequence in reverse order.

```
list(reversed(range(10))) *
```

* reversed() returns the iterator, list() makes it a list.

Control and Flow

1. Operators for conditions in 'if else' :

Check if two variables are same object	`var1 is var2`
... are different object	`var1 is not var2`
Check if two variables have same value	`var1 == var2`

WARNING : Use 'and', 'or', 'not' operators for compound conditions, not &&, \|\|, !.

2. Common usage of 'for' operator :

Iterating over a collection (i.e. list or tuple) or an iterator	`for element in iterator :`
... If elements are sequences, can be 'unpack'	`for a, b, c in iterator :`

3. 'pass' - no-op statement. Used in blocks where no action is to be taken.
4. Ternary Expression - aka less verbose 'if else'
   - Basic Form :

```
value = true-expr if condition
else false-expr
```

5. No switch/case statement, use if/elif instead.

Object-Oriented Programming

1. **'object'** is the root of all Python types
2. Everything (number, string, function, class, module, etc.) is an object, each object has a 'type'. Object variable is a pointer to its location in memory.
3. All objects are reference-counted.

```
sys.getrefcount(5)  =>  x
```

```
a = 5, b = a
# This creates a 'reference' to the object on the right side of =, thus both a and b point to 5
```

```
sys.getrefcount(5)  => x + 2
```

```
del(a); sys.getrefcount(5) => x + 1
```

4. **Class** Basic Form :

```
class MyObject(object):
      # 'self' is equivalent of 'this' in Java/C++
      def __init__(self, name):
          self.name = name
      def memberFunc1(self, arg1):
          ..
      @staticmethod
      def classFunc2(arg1):
          ..
obj1 = MyObject('name1')
obj1.memberFunc1('a')
MyObject.classFunc2('b')
```

5. Useful interactive tool :

```
dir(variable1) # list all methods available on the object
```

Common String OPERATIONS

Concatenate List/Tuple with Separator	`', '.join([ 'v1', 'v2', 'v3']) =>  'v1, v2, v3'`

Format String	`string1 = 'My name is {0} {name}'`
	`newString1 = string1.format('Sean', name = 'Chen')`

Split String	`sep = '-';`
	`stringList1 = string1.split(sep)`

Get Substring	`start = 1;  string1[start:8]`

String Padding with Zeros	`month = '5';`
	`month.zfill(2) => '05'`
	`month = '12';`
	`month.zfill(2) => '12'`

Exception Handling

1. Basic Form :

```
try:
      ..
except ValueError as e:
      print e
except (TypeError, AnotherError):
      ..
except:
      ..
finally:
      .. # clean up, e.g. close db
```

2. Raise Exception Manually

```
raise AssertionError  # assertion failed
raise SystemExit      # request program exit
raise RuntimeError('Error message :
..')
```

List, Set and Dict Comprehansions

Syntactic sugar that makes code easier to read and write

1. **List comprehensions**
   - Concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression.
   - Basic form :

```
[expr for val in collection if condition]
```

   A shortcut for :

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

   The filter condition can be omitted, leaving only the expression.

2. **Dict Comprehension**
   - Basic form :

```
{key-expr : value-expr for value in
collection if condition}
```

3. **Set Comprehension**
   - Basic form : same as List Comprehension except with curly braces instead of []
4. **Nested list Comprehensions**
   - Basic form :

```
[expr for val in collection for
innerVal in val if condition]
```

Created by Arianne Colton and Sean Chen
data.scientist.info@gmail.com

Based on content from
'Python for Data Analysis' by Wes McKinney

Updated: May 3, 2016