**Python Notes/Cheat Sheet**

**Comments**

# from the hash symbol to the end of a line

**Code blocks**

Delineated by colons and indented code; and not the curly brackets of C, C++ and Java.

```python
def is_fish_as_string(argument):
    if argument:
        return 'fish'
    else:
        return 'not fish'
```

**Note**: Four spaces per indentation level is the Python standard. Never use tabs: mixing tabs and spaces produces hard-to-find errors. Set your editor to convert tabs to spaces.

**Line breaks**

Typically, a statement must be on one line. Bracketed code - (), [], or {} - can be split across lines; or (if you must) use a backslash \ at the end of a line to continue a statement on to the next line (but this can result in hard to debug code).

**Naming conventions**

<table>
<thead>
<tr>
<th>Style</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>StudlyCase</td>
<td>Class names</td>
</tr>
<tr>
<td>joined_lower</td>
<td>Identifiers, functions; and class methods, attributes</td>
</tr>
<tr>
<td>__joined_lower</td>
<td>Internal class attributes</td>
</tr>
<tr>
<td>__joined_lower</td>
<td>Private class attributes # this use not recommended</td>
</tr>
<tr>
<td>joined_lower</td>
<td>Constants</td>
</tr>
<tr>
<td>ALL_CAPS</td>
<td></td>
</tr>
</tbody>
</table>

**Basic object types (not a complete list)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None             # singleton null object</td>
</tr>
<tr>
<td>Boolean</td>
<td>True, False</td>
</tr>
<tr>
<td>integer</td>
<td>-1, 0, 1, sys.maxint</td>
</tr>
<tr>
<td>long</td>
<td>1L, 9787L # arbitrary length ints</td>
</tr>
<tr>
<td>float</td>
<td>3.14159265 # infinity</td>
</tr>
<tr>
<td>complex</td>
<td>2+3j               # note use of j</td>
</tr>
</tbody>
</table>
| string    | 'i am a string', "me too"
|           | "multi-line string", "..." |
|           | r'raw string', b'raw string', u'unicode string' |
| tuple     | empty = () # empty tuple |
|           | (1, True, 'dog') # immutable list |
| list      | empty = [] # empty list |
|           | [1, True, 'dog') # mutable list |
| set       | empty = set() # empty set |
|           | set(1, True, 'a') # mutable |
| dictionary| empty = {} # mutable object |
|           | {'a': 'dog', 7: 'seven', True: 1} |
| file      | f = open('filename', 'rb') |

**Note**: Python has four numeric types (integer, float, long and complex) and several sequence types including strings, lists, tuples, bytearrays, buffers, and xrange objects.

**Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition (also string, tuple, list concatenation)</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction (also set difference)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication (also string, tuple, list replication)</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (also a string format function, but use deprecated)</td>
</tr>
<tr>
<td>//</td>
<td>Integer division rounded towards minus infinity</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>=, -=, +=, /=, *=, %=, //=, **=</td>
<td>Assignment operators</td>
</tr>
<tr>
<td>and, or, not</td>
<td>Boolean operators</td>
</tr>
<tr>
<td>in, not in</td>
<td>Membership test operators</td>
</tr>
<tr>
<td>is, is not</td>
<td>Object identity operators</td>
</tr>
<tr>
<td></td>
<td>Left and right bit shift</td>
</tr>
<tr>
<td>;</td>
<td>Inline statement separator # inline statements discouraged</td>
</tr>
</tbody>
</table>

**Hint**: float('inf') always tests as larger than any number, including integers.

**Modules**

Modules open up a world of Python extensions that can be imported and used. Access to the functions, variables and classes of a module depend on how the module was imported.

<table>
<thead>
<tr>
<th>Import method</th>
<th>Access/Use syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>import math</td>
<td>math.cos(math.pi/3)</td>
</tr>
<tr>
<td>import math as m</td>
<td>m.cos(m.pi/3)</td>
</tr>
<tr>
<td># import using an alias</td>
<td></td>
</tr>
<tr>
<td>from math import cos,pi</td>
<td>cos(pi/3)</td>
</tr>
<tr>
<td># only import specifics</td>
<td></td>
</tr>
<tr>
<td>from math import *</td>
<td>log(e)</td>
</tr>
</tbody>
</table>

Global imports make for unreadable code!!!

**Oft used modules**

<table>
<thead>
<tr>
<th>Module</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td>Date and time functions</td>
</tr>
<tr>
<td>time</td>
<td></td>
</tr>
<tr>
<td>math</td>
<td>Core math functions and the constants pi and e</td>
</tr>
<tr>
<td>pickle</td>
<td>Serialise objects to a file</td>
</tr>
<tr>
<td>os</td>
<td>Operating system interfaces</td>
</tr>
<tr>
<td>os.path</td>
<td></td>
</tr>
<tr>
<td>re</td>
<td>A library of Perl-like regular expression operations</td>
</tr>
<tr>
<td>string</td>
<td>Useful constants and classes</td>
</tr>
<tr>
<td>sys</td>
<td>System parameters and functions</td>
</tr>
<tr>
<td>numpy</td>
<td>Numerical constants and classes</td>
</tr>
<tr>
<td>pandas</td>
<td>R DataFrames for Python</td>
</tr>
<tr>
<td>matplotlib</td>
<td>Plotting/charting for Python</td>
</tr>
</tbody>
</table>

Version 14 March 2015 – [Draft – Mark Graph – mark dot the dot graph at gmail dot com – @Mark_Graph on twitter]
If - flow control

```
if condition:  # for example: if x < 5:
    statements
elif condition:  # optional – can be multiple
    statements
else:  # optional
    statements
```

For - flow control

```
for x in iterable:
    statements
else:  # optional completion code
    statements
```

While - flow control

```
while condition:
    statements
else:  # optional completion code
    statements
```

Ternary statement

```
id = expression if condition else expression
```

```
x = y if a > b else z - 5
```

Some useful adjuncts:
- pass - a statement that does nothing
- continue - moves to the next loop iteration
- break - to exit for and while loop

**Trap:** break skips the else completion code

Exceptions – flow control

```
try:
    statements
except (tuple_of_errors):  # can be multiple
    statements
else:
    statements
finally:
    statements
```

Common exceptions (not a complete list)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Why it happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssertionError</td>
<td>Assert statement failed</td>
</tr>
<tr>
<td>AttributeError</td>
<td>Class attribute assignment or reference failed</td>
</tr>
<tr>
<td>IOError</td>
<td>Failed I/O operation</td>
</tr>
<tr>
<td>ImportError</td>
<td>Failed module import</td>
</tr>
<tr>
<td>IndexError</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>KeyError</td>
<td>Dictionary key not found</td>
</tr>
<tr>
<td>MemoryError</td>
<td>Ran out of memory</td>
</tr>
<tr>
<td>NameError</td>
<td>Name not found</td>
</tr>
<tr>
<td>TypeError</td>
<td>Value of the wrong type</td>
</tr>
<tr>
<td>ValueError</td>
<td>Right type but wrong value</td>
</tr>
</tbody>
</table>

Raising errors

```
Errors are raised using the raise statement

raise ValueError(value)
```

Creating new errors

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

Objects and variables (AKA identifiers)

- Everything is an object in Python (in the sense that it can be assigned to a variable or passed as an argument to a function)
- Most Python objects have methods and attributes. For example, all functions have the built-in attribute `__doc__`, which returns the doc string defined in the function's source code.
- All variables are effectively "pointers", not "locations". They are references to objects; and often called identifiers.
- Objects are strongly typed, not identifiers
- Some objects are immutable (int, float, string, tuple, frozenset). But most are mutable (including: list, set, dictionary, NumPy arrays, etc.)
- You can create your own object types by defining a new class (see below).

**Booleans and truthiness**

Most Python objects have a notion of "truth".

<table>
<thead>
<tr>
<th>False</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Any number other than 0</td>
</tr>
<tr>
<td>int(False)</td>
<td># other number</td>
</tr>
<tr>
<td>***</td>
<td>&quot;&quot; , 'fred', 'False'</td>
</tr>
<tr>
<td># the empty string</td>
<td># all other strings</td>
</tr>
<tr>
<td>() [] {} set()</td>
<td>[None], (False), {1, 1}</td>
</tr>
<tr>
<td># empty containers</td>
<td># non-empty containers, including those containing False or None.</td>
</tr>
</tbody>
</table>

You can use bool() to discover the truth status of an object.

```
a = bool(obj)          # the truth of obj
```

It is pythonic to use the truth of objects.

```
if container:
    # test not empty
    # do something
while items:
    # common looping idiom
    item = items.pop()
    # process item
```

Specify the truth of the classes you write using the `__nonzero__()` magic method.

**Comparisons**

Python lets you compare ranges, for example

```
if 1 < x < 100:  # do something ...
```

**Tuples**

Tuples are immutable lists. They can be searched, indexed and iterated much like lists (see below). List methods that do not change the list also work on tuples.

```
a = ()               # the empty tuple
a = (1,)  # note comma # one item tuple
a = (1, 2, 3)    # multi-item tuple
a = ((1, 2), (3, 4))   # nested tuple
a = tuple(['a', 'b']) # conversion
```

Note: the comma is the tuple constructor, not the parentheses. The parentheses add clarity.

**The Python swap variable idiom**

```
a, b = b, a  # no need for a temp variable
```

This syntax uses tuples to achieve its magic.
String (immutable, ordered, characters)

s = "string".upper() # STRING
s = 'fred'+'was'+'here' # concatenation
d = " '.join(['fred', 'was', 'here']) # ditto
s = 'spam' * 3 # replication
s = str(x) # conversion

String iteration and sub-string searching

for character in "str'": # iteration
    print (ord(character)) # 115 116 114
for index, character in enumerate("str'"): # 0 1 2
    print (index, character) # 0 's' 1 't' 2 'r'
if 'red' in "Fred": # searching
    print ('Fred is red') # it prints!
if 'red' in 'Fred': # searching
    print ('Fred is red') # it prints!

String methods (not a complete list)

capitalize, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

String constants (not a complete list)

from string import * # I'm bad ...
print ("{}", format(math.pi)) # 3.14159265359
print ("{}", format(math.pi)) # 3.141593
print ("{}", format(3000)) # 3.00e+03
print ("{}", format(1000000)) # 1,000,000

Old school string formatting (using % oper)

print("It %d times", % ['occurred', 5]) # prints: 'It occurred 5 times'

Old new string formatting (using format method)

Example: using similar codes as above:

'Hello'.format("World") # 'Hello World'
"{}".format(math.pi) # 3.14159265359
"{}", format(math.pi) # 3.141593
"{}", format(3000) # 3.00e+03
"{}", format(1000000) # 1,000,000

New string formatting (using format method)

Uses: 'template-string'.format(arguments)

Examples (using similar codes as above):

x = [0, 1, 2, 3, 4, 5, 6, 7, 8] # play data
x[2] # 3rd element - reference not slice
x[1:3] # 2nd to 3rd element (1, 2)
x[:3] # The first three elements (0,1,2)
x[-3:] # last three elements
x[:-3] # all but the last three elements
x[:] # every element of x - copies x
x[1:-1] # all but first and last element
x[:3] # (0, 3, 6, 9, ...) 1st then every 3rd
x[1:5:2] # (1,3) start 1, stop >= 5, every 2

String (immutable, ordered, characters)

s = "string".upper() # STRING
s = 'fred'+'was'+'here' # concatenation
d = " '.join(['fred', 'was', 'here']) # ditto
s = 'spam' * 3 # replication
s = str(x) # conversion

String iteration and sub-string searching

for character in "str'": # iteration
    print (ord(character)) # 115 116 114
for index, character in enumerate("str'"): # 0 1 2
    print (index, character) # 0 's' 1 't' 2 'r'
if 'red' in "Fred": # searching
    print ('Fred is red') # it prints!
if 'red' in 'Fred': # searching
    print ('Fred is red') # it prints!

String methods (not a complete list)

capitalize, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

String constants (not a complete list)

from string import * # I'm bad ...
print ("{}", format(math.pi)) # 3.14159265359
print ("{}", format(math.pi)) # 3.141593
print ("{}", format(3000)) # 3.00e+03
print ("{}", format(1000000)) # 1,000,000

Old school string formatting (using % oper)

print("It %d times", % ['occurred', 5]) # prints: 'It occurred 5 times'

Old new string formatting (using format method)

Example: using similar codes as above:

'Hello'.format("World") # 'Hello World'
"{}".format(math.pi) # 3.14159265359
"{}", format(math.pi) # 3.141593
"{}", format(3000) # 3.00e+03
"{}", format(1000000) # 1,000,000

New string formatting (using format method)

Uses: 'template-string'.format(arguments)

Examples (using similar codes as above):

'Hello'.format("World") # 'Hello World'
"{}".format(math.pi) # 3.14159265359
"{}", format(math.pi) # 3.141593
"{}", format(3000) # 3.00e+03
"{}", format(1000000) # 1,000,000
Set (unique, unordered container)
A Python set is an unordered, mutable collection of unique hashable objects.

```python
da = set()             # empty set
da = {'red', 'white', 'blue'}  # simple set
da = set(x)             # convert list
f = frozenset(o)             # convert other
f = frozenset(s)             # convert set
therefore hashable).
```

**Frozenset**

- `s.difference(o)`
- `s.intersection(o)`
- `s.union(o[, ...])`
- `s.issuperset(o)`
- `s.issubset(o)`
- `s.copy()`
- `isdisjoint(o)`

**Set methods (not a complete list)**

<table>
<thead>
<tr>
<th>Method</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td>Number of items in set</td>
</tr>
<tr>
<td><code>s.add(item)</code></td>
<td>Add item to set</td>
</tr>
<tr>
<td><code>s.remove(item)</code></td>
<td>Remove item from set. Raise KeyError if item not found.</td>
</tr>
<tr>
<td><code>s.discard(item)</code></td>
<td>Remove item from set if present.</td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td>Remove and return an arbitrary item. Raise KeyError on empty set.</td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td>Remove all items from set</td>
</tr>
<tr>
<td><code>item in s</code></td>
<td>True or False</td>
</tr>
<tr>
<td><code>item not in s</code></td>
<td>True or False</td>
</tr>
<tr>
<td><code>iter(s)</code></td>
<td>An iterator over the items in the set (arbitrary order)</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td>Get shallow copy of set</td>
</tr>
<tr>
<td><code>s.isdisjoint(o)</code></td>
<td>True if s has not items in common with other set o</td>
</tr>
<tr>
<td><code>s.issubset(o)</code></td>
<td>Same as s &lt;= other</td>
</tr>
<tr>
<td><code>s.issuperset(o)</code></td>
<td>Same as s =&gt; other</td>
</tr>
<tr>
<td><code>s.union(o[, ...])</code></td>
<td>Return new union set</td>
</tr>
<tr>
<td><code>s.intersection(o)</code></td>
<td>Return new intersection</td>
</tr>
<tr>
<td><code>s.difference(o)</code></td>
<td>Get net set of items in s but not others (Same as set – other)</td>
</tr>
</tbody>
</table>

**Dictionary (indexed, unordered map-container)**
A mutable hash map of unique key=value pairs.

```python
a = {}             # empty dictionary
a = {1: 1, 2: 4, 3: 9}  # simple dict
a = dict(x) # convert paired data
# next example – create from a list
l = ['alpha', 'beta', 'gamma', 'delta']
a = dict(zip(range(len(l)), l))
# Example using string & generator expression
s = 'apple,bird,cat,dog,egg'
a = dict(i.split(',') for i in s.split(','))
```

**Dictionary comprehensions**

Conceptually like list comprehensions; but it constructs a dictionary rather than a list.

```python
a = { n: n*n for n in range(7) }
# a -> {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}
odd_sq = { n: n*n for n in range(7) if n%2 }
# odd_sq -> {1: 1, 3: 9, 5: 25}
# next example -> swaps the key:value pairs
a = { val: key for key, val in a.items() }  
# next example -> count list occurrences
l = [1,2,9,2,7,3,7,1,22,1,7,7,22,22,9,0,9,0]
c = { key: l.count(key) for key in set(l) }
# c -> {0:2, 1:3, 2:2, 3:1, 7:4, 9:3, 22:3}
```

**Iterating a dictionary**

```python
for key in dictionary:
    print (key)
for key, value in dictionary.items():
    print (key, value)
```

**Searching a dictionary**

```python
if key in dictionary:
    print (key)
```

**Merging two dictionaries**

```python
merged = dict_1.copy()
merged.update(dict_2)
```

**Dictionary methods (not a complete list)**

<table>
<thead>
<tr>
<th>Method</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(d)</code></td>
<td>Number of items in d</td>
</tr>
<tr>
<td><code>d[key]</code></td>
<td>Get value for key or raise the KeyError exception</td>
</tr>
<tr>
<td><code>d[key] = value</code></td>
<td>Set key to value</td>
</tr>
<tr>
<td><code>del d[key]</code></td>
<td>deletion</td>
</tr>
<tr>
<td><code>key in d</code></td>
<td>True or False</td>
</tr>
<tr>
<td><code>key not in d</code></td>
<td>True or False</td>
</tr>
<tr>
<td><code>iter(d)</code></td>
<td>An iterator over the keys</td>
</tr>
<tr>
<td><code>d.clear()</code></td>
<td>Remove all items from d</td>
</tr>
<tr>
<td><code>d.copy()</code></td>
<td>Shallow copy of dictionary</td>
</tr>
<tr>
<td><code>d.get(key[, def])</code></td>
<td>Get value else default</td>
</tr>
<tr>
<td><code>d.items()</code></td>
<td>Dictionary's (k,v) pairs</td>
</tr>
<tr>
<td><code>d.keys()</code></td>
<td>Dictionary's keys</td>
</tr>
<tr>
<td><code>d.pop(key[, def])</code></td>
<td>Get value else default; remove key from dictionary</td>
</tr>
<tr>
<td><code>d.popitem()</code></td>
<td>Remove and return an arbitrary (k, v) pair</td>
</tr>
<tr>
<td><code>d.setdefault(key[, def])</code></td>
<td>If k in dict return its value otherwise set def</td>
</tr>
<tr>
<td><code>d.update(other_d)</code></td>
<td>Update d with key:val pairs from other</td>
</tr>
<tr>
<td><code>d.values()</code></td>
<td>The values from dict</td>
</tr>
</tbody>
</table>

**Frozenset**

Similar to a Python set above, but immutable (and therefore hashable).

```python
f = frozenset(s)             # convert set
f = frozenset(o)             # convert other
```
### Key functions (not a complete list)

<table>
<thead>
<tr>
<th>Function</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(num)</td>
<td>Absolute value of num</td>
</tr>
<tr>
<td>all(iterable)</td>
<td>True if all are True</td>
</tr>
<tr>
<td>any(iterable)</td>
<td>True if any are True</td>
</tr>
<tr>
<td>bytearray(source)</td>
<td>A mutable array of bytes</td>
</tr>
<tr>
<td>callable(obj)</td>
<td>True if obj is callable</td>
</tr>
<tr>
<td>chr(int)</td>
<td>Character for ASCII int</td>
</tr>
<tr>
<td>complex(re[, im])</td>
<td>Create a complex number</td>
</tr>
<tr>
<td>divmod(a, b)</td>
<td>Get (quotient, remainder)</td>
</tr>
<tr>
<td>enumerate(seq)</td>
<td>Get an enumerate object, with next() method returns an (index, element) tuple</td>
</tr>
<tr>
<td>eval(string)</td>
<td>Evaluate an expression</td>
</tr>
<tr>
<td>filter(fn, iter)</td>
<td>Construct a list of elements from iter for which fn() returns True</td>
</tr>
<tr>
<td>float(x)</td>
<td>Convert from int/string</td>
</tr>
<tr>
<td>getattr(obj, str)</td>
<td>Like obj.str</td>
</tr>
<tr>
<td>hasattr(obj, str)</td>
<td>True if obj has attribute</td>
</tr>
<tr>
<td>hex(x)</td>
<td>From in to hex string</td>
</tr>
<tr>
<td>id(obj)</td>
<td>Return unique (run-time) identifier for an object</td>
</tr>
<tr>
<td>int(x)</td>
<td>Convert from float/string</td>
</tr>
<tr>
<td>isinstance(o, c)</td>
<td>Eg. isinstance(2.1, float)</td>
</tr>
<tr>
<td>len(x)</td>
<td>Number of items in x; x is string, tuple, list, dict</td>
</tr>
<tr>
<td>list(iterable)</td>
<td>Make a list</td>
</tr>
<tr>
<td>long(x)</td>
<td>Convert a string or number to a long integer</td>
</tr>
<tr>
<td>map(fn, iterable)</td>
<td>Apply fn() to every item in iterable; return results in a list</td>
</tr>
<tr>
<td>max(a,b)</td>
<td>What it says on the tin</td>
</tr>
<tr>
<td>maxiterable</td>
<td>Ditto</td>
</tr>
<tr>
<td>min(a,b)</td>
<td>Ditto</td>
</tr>
<tr>
<td>miniterable</td>
<td>Ditto</td>
</tr>
<tr>
<td>next(iterator)</td>
<td>Get next item from an iter</td>
</tr>
<tr>
<td>open(name[,mode])</td>
<td>Open a file object</td>
</tr>
<tr>
<td>ord(c)</td>
<td>Opposite of chr(int)</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>Same as x ** y</td>
</tr>
<tr>
<td>print (objects)</td>
<td>What it says on the tin</td>
</tr>
<tr>
<td>range(stop)</td>
<td>integer list; stops &lt; stop</td>
</tr>
<tr>
<td>range(start,stop)</td>
<td>default start=0; default step=1</td>
</tr>
<tr>
<td>range(fr,to,step)</td>
<td>Applies the two argument fn(x, y) cumulatively to the items of iter.</td>
</tr>
<tr>
<td>reduce(fn, iter)</td>
<td>Print a reversed representation of an object</td>
</tr>
<tr>
<td>repr(object)</td>
<td>cumulatively to the items of iter.</td>
</tr>
<tr>
<td>reversed(seq)</td>
<td>Get a reversed iterator</td>
</tr>
<tr>
<td>round(n[,digits])</td>
<td>Round to number of digits after the decimal place</td>
</tr>
<tr>
<td>setattr(obj,n,v)</td>
<td>Like obj._n = v #name/value</td>
</tr>
<tr>
<td>sorted(iterable)</td>
<td>Get new sorted list</td>
</tr>
<tr>
<td>str(object)</td>
<td>Get a string for an object</td>
</tr>
<tr>
<td>sum(iterable)</td>
<td>Sum list of numbers</td>
</tr>
<tr>
<td>type(object)</td>
<td>Get the type of object</td>
</tr>
<tr>
<td>xrange()</td>
<td>Like range() but better: returns an iterator</td>
</tr>
<tr>
<td>zip(x, y[, z])</td>
<td>Return a list of tuples</td>
</tr>
</tbody>
</table>

### Using functions

When called, functions can take positional and named arguments.

For example:

```python
result = function(32, aVar, c='see', d={})
```

Arguments are passed by reference (ie. the objects are not copied, just the references).

### Writing a simple function

```python
def funct(arg1, arg2=None, *args, **kwargs):
    """explain what this function does"
    statements
    return x  # optional statement
```

Note: functions are first class objects that get instantiated with attributes and they can be referenced by variables.

### Avoid named default mutable arguments

Avoid mutable objects as default arguments. Expressions in default arguments are evaluated when the function is defined, not when it's called. Changes to mutable default arguments survive between function calls.

```python
def nasty(value=[]):
    # <-- mutable arg
    value.append('a')
    return value
```

```python
print (nasty ()) # --> [ 'a' ]
print (nasty ()) # --> [ 'a', 'a' ]
```

```python
def better(val=None):
    val = [] if val is None else val
    value.append('a')
    return value
```

### Lambda (inline expression) functions:

```python
g = lambda x: x ** 2  # Note: no return
print (g(8))  # prints 64
mul = lambda a, b: a * b  # two arguments
mul(4, 5) == 4 * 5  # --> True
```

Note: only for expressions, not statements. Lambdas are often used with the Python functions filter(), map() and reduce().

```python
# get only those numbers divisible by three
div3 = filter(lambda x: x%3==0,range(1,101))
```

Typically, you can put a lambda function anywhere you put a normal function call.

### Closures

Closures are functions that have inner functions with data fixed in the inner function by the lexical scope of the outer. They are useful for avoiding hard constants. Wikipedia has a derivative function for changeable values of dx, using a closure.

```python
def derivative(f, dx):
    """Return a function that approximates the derivative of f using an interval of dx, which should be appropriately small."
    _function(x):
        return (f(x + dx) - f(x)) / dx
    return _function #from derivative(f, dx)
```

```python
f_dash_x = derivative(lambda x: x**x,0.000001)
f_dash_x(5) # yields approx. 10 (ie. y'=2x)
```
An iterable object
The contents of an iterable object can be selected one at a time. Such objects include the Python sequence types and classes with the magic method __iter__(), which returns an iterator. An iterable object will produce a fresh iterator with each call to iter().

```
iterator = iter(iterable_object)
```

Iterators
Objects with a next() (Python 2) or __next__() (Python 3) method, that:
• returns the next value in the iteration
• updates the internal note of the next value
• raises a StopIteration exception when done

Note: with the loop for x in y: if y is not an iterator; Python calls iter() to get one. With each loop, it calls next() on the iterator until a StopIteration exception.

```
x = iter('XY') # iterate a string by hand
print (next(x)) # --> X
print (next(x)) # --> Y
print (next(x)) # --> StopIteration
```

Generators
Generator functions are resumable functions that work like iterators. They can be more space or time efficient than iterating over a list, (especially a very large list), as they only produce items as they are needed.

```
def fib(max=None):
    """ generator for Fibonacci sequence"""  
    a, b = 0, 1
    while max is None or b <= max:
        yield b   # yield is like return
        a, b = b, a+b
    [i for i in fib(10)] # --> [1, 1, 2, 3, 5, 8]
```

Note: a return statement (or getting to the end of the function) ends the iteration.
Trap: a yield statement is not allowed in the try clause of a try/finally construct.

Messaging the generator
```
def resetableCounter(max=None):
    j = 0
    while max is None or j <= max:
        x = yield j   # x gets the sent arg
        j = j+1 if x is None else x

x = resetableCounter(10)
print x.send(None) # --> 0
print x.send(5)    # --> 5
print x.send(None) # --> 6
print x.send(11)   # --> StopIteration
```

Trap: must send None on first send() call

Generator expressions
Generator expressions build generators, just like building a list from a comprehension. You can turn a list comprehension into a generator expression simply by replacing the square brackets [] with parentheses ().

```
[i for i in xrange(10)] # list comprehension
list(i for i in xrange(10)) # generated list
```

Classes
Python is an object-oriented language with a multiple inheritance class mechanism that encapsulates program code and data.

Methods and attributes
Most objects have associated functions or "methods" that are called using dot syntax:
```
obj.method(arg)
```

Objects also often have attributes or values that are directly accessed without using getters and setters (most unlike Java or C++)
```
instance = Example_Class()
print (instance.attribute)
```

Simple example
```
import math

class Point:
    # static class variable, point count
    count = 0

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
        Point.count += 1

    def __str__(self):
        return \n            '(x={}, y={})'.format(self.x, self.y)

    def to_polar(self):
        r = math.sqrt(self.x**2 + self.y**2)
        theta = math.atan2(self.y, self.x)
        return (r, theta)

# static method - trivial example ...

def static_eg(n):
    print ('{}'.format(n))
static_eg = staticmethod(static_eg)

# Instantiate 9 points & get polar coords
for x in range(-1, 2):
    for y in range(-1, 2):
        p = Point(x, y)
        print (p)   # uses __str__() method
        print (p.to_polar())
Point.static_eg(9)   # check static variable
```

The self
Class methods have an extra argument over functions. Usually named 'self', it is a reference to the instance. It is not used in the method call; and is provided by Python to the method. Self is like 'this' in C++ & Java

Public and private methods and variables
Python does not enforce the public v private data distinction. By convention, variables and methods that begin with an underscore should be treated as private (unless you really know what you are doing). Variables that begin with double underscore are mangled by the compiler (and hence more private).
Inheritance

class DerivedClass1(BaseClass):
    statements

class DerivedClass2(module_name.BaseClass):
    statements

Multiple inheritance

class DerivedClass(Base1, Base2, Base3):
    statements

Decorators

Technically, decorators are just functions (or classes), that take a callable object as an argument, and return an analogous object with the decoration. We will skip how to write them, and focus on using a couple of common built in decorators.

Practically, decorators are syntactic sugar for more readable code. The @wrapper is used to transform the existing code. For example, the following two method definitions are semantically equivalent.

def f(...):
    ...

f = staticmethod(f)

@staticmethod
def f(...):
    ...

Getters and setters

Although class attributes can be directly accessed, the property function creates a property manager.

class Example:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx,"Doc txt")

Which can be rewritten with decorators as:

class Example:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """Doc txt: I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

Magic class methods (not a complete list)

Magic methods (which begin and end with double underscore) add functionality to your classes consistent with the broader language.

<table>
<thead>
<tr>
<th>Magic method</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(self,...)</td>
<td>Constructor</td>
</tr>
<tr>
<td><strong>del</strong>(self)</td>
<td>Destructor pre-garbage collection</td>
</tr>
<tr>
<td><strong>str</strong>(self)</td>
<td>Human readable string for class contents. Called by str(self)</td>
</tr>
<tr>
<td><strong>repr</strong>(self)</td>
<td>Machine readable unambiguous Python string expression for class contents. Called by repr(self) Note: str(self) will call <strong>repr</strong> if <strong>str</strong> is not defined.</td>
</tr>
<tr>
<td><strong>eq</strong>(self, other)</td>
<td>Behaviour for ==</td>
</tr>
<tr>
<td><strong>ne</strong>(self, other)</td>
<td>Behaviour for !=</td>
</tr>
<tr>
<td><strong>lt</strong>(self, other)</td>
<td>Behaviour for &lt;</td>
</tr>
<tr>
<td><strong>gt</strong>(self, other)</td>
<td>Behaviour for &gt;</td>
</tr>
<tr>
<td><strong>le</strong>(self, other)</td>
<td>Behaviour for &lt;=</td>
</tr>
<tr>
<td><strong>ge</strong>(self, other)</td>
<td>Behaviour for =&gt;</td>
</tr>
<tr>
<td><strong>add</strong>(self, other)</td>
<td>Behaviour for +</td>
</tr>
<tr>
<td><strong>sub</strong>(self, other)</td>
<td>Behaviour for -</td>
</tr>
<tr>
<td><strong>mul</strong>(self, other)</td>
<td>Behaviour for *</td>
</tr>
<tr>
<td><strong>div</strong>(self, other)</td>
<td>Behaviour for /</td>
</tr>
<tr>
<td><strong>mod</strong>(self, other)</td>
<td>Behaviour for %</td>
</tr>
<tr>
<td><strong>pow</strong>(self, other)</td>
<td>Behaviour for **</td>
</tr>
<tr>
<td><strong>pos</strong>(self, other)</td>
<td>Behaviour for unary +</td>
</tr>
<tr>
<td><strong>neg</strong>(self, other)</td>
<td>Behaviour for unary -</td>
</tr>
<tr>
<td><strong>hash</strong>(self)</td>
<td>Returns an int when hash() called. Allows class instance to be put in a dictionary</td>
</tr>
<tr>
<td><strong>len</strong>(self)</td>
<td>Length of container</td>
</tr>
<tr>
<td><strong>contains</strong>(self, i)</td>
<td>Behaviour for in and not in operators</td>
</tr>
<tr>
<td><strong>missing</strong>(self, i)</td>
<td>What to do when dict key is missing</td>
</tr>
<tr>
<td><strong>copy</strong>(self)</td>
<td>Shallow copy constructor</td>
</tr>
<tr>
<td><strong>deepcopy</strong>(self, memodict={})</td>
<td>Deep copy constructor</td>
</tr>
<tr>
<td><strong>iter</strong>(self)</td>
<td>Provide an iterator</td>
</tr>
<tr>
<td><strong>nonzero</strong>(self)</td>
<td>Called by bool(self)</td>
</tr>
<tr>
<td><strong>index</strong>(self)</td>
<td>Called by x[self]</td>
</tr>
<tr>
<td><strong>setattr</strong>(self, name, val)</td>
<td>Called by self.name = val</td>
</tr>
<tr>
<td><strong>getattribute</strong>(self, name)</td>
<td>Called by self.name</td>
</tr>
<tr>
<td><strong>getattr</strong>(self, name)</td>
<td>Called when self.name does not exist</td>
</tr>
<tr>
<td><strong>delattr</strong>(self, name)</td>
<td>Called by del self.name</td>
</tr>
<tr>
<td><strong>getitem</strong>(self, key)</td>
<td>Called by self[key]</td>
</tr>
<tr>
<td><strong>setitem</strong>(self, key, val)</td>
<td>Called by self[key] = val</td>
</tr>
<tr>
<td><strong>delitem</strong>(self, key)</td>
<td>del self[key]</td>
</tr>
</tbody>
</table>