

# 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.

```
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

Style	Use
<b>StudyCase</b>	Class names
<b>joined_lower</b>	Identifiers, functions; and class methods, attributes
<b>_joined_lower</b>	Internal class attributes
<b>__joined_lower</b>	Private class attributes # this use not recommended
<b>joined_lower</b>	Constants
<b>ALL_CAPS</b>	

## Basic object types (not a complete list)

Type	Examples
<b>None</b>	None # singleton null object
<b>Boolean</b>	True, False
<b>integer</b>	-1, 0, 1, sys.maxint
<b>long</b>	1L, 9787L # arbitrary length ints
<b>float</b>	3.14159265 inf, float('inf') # infinity -inf # neg infinity nan, float('nan') # not a number
<b>complex</b>	2+3j # note use of j
<b>string</b>	'I am a string', "me too" """multi-line string", """+1""" r'raw string', b'ASCII string' u'unicode string'
<b>tuple</b>	empty = () # empty tuple (1, True, 'dog') # immutable list
<b>list</b>	empty = [] # empty list [1, True, 'dog'] # mutable list
<b>set</b>	empty = set() # the empty set set(1, True, 'a') # mutable
<b>dictionary</b>	empty = {} # mutable object {'a': 'dog', 7: 'seven', True: 1}
<b>file</b>	f = open('filename', 'rb')

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

## Operators

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

**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.

Import method	Access/Use syntax
<b>import math</b>	math.cos(math.pi/3)
<b>import math as m</b> # import using an alias	m.cos(m.pi/3)
<b>from math import cos, pi</b> # only import specifics	cos(pi/3)
<b>from math import *</b> # <b>BADish</b> global import	log(e)

Global imports make for unreadable code!!!

## Oft used modules

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

## 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: # optional no exceptions
    statements
finally: # optional all
    statements
```

## Common exceptions (not a complete list)

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

## 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 our own object types by defining a new class (see below).

## Booleans and truthiness

Most Python objects have a notion of "truth".

False	True
None	
0	Any number other than 0
int(False) # → 0	int(True) # → 1
""	" ", 'fred', 'False'
# the empty string	# all other strings
() [] {} set()	[None], (False), {1, 1}
# empty containers	# non-empty containers, including those containing False or None.

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
s = ''.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')
    print (index, character)
if 'red' in 'Fred':      # searching
    print ('Fred is red') # it prints!
```

## String methods (not a complete list)

capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, 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 ((digits, hexdigits, letters,
        lowercase, uppercase, punctuation))
```

## Old school string formatting (using % oper)

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

Code	Meaning
s	String or string conversion
c	Character
d	Signed decimal integer
u	Unsigned decimal integer
H or h	Hex integer (upper or lower case)
f	Floating point
E or e	Exponent (upper or lower case E)
G or g	The shorter of e and f (u/l case)
%	Literal '%'

```
'%s' % math.pi      # --> '3.14159265359'
'%f' % math.pi      # --> '3.141593'
'%0.2f' % math.pi   # --> '3.14'
'%0.2e' % 3000      # --> '3.00e+03'
'%03d' % 5          # --> '005'
```

## 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'
'{0:.2f}'.format(math.pi)   # '3.14'
'{0:+.2f}'.format(5)        # '+5.00'
'{:.2e}'.format(3000)       # '3.00e+03'
'{:0>2d}'.format(5)         # '05' (left pad)
'{:x<3d}'.format(5)         # '5xx' (rt. pad)
'{:,}'.format(1000000)      # '1,000,000'
'{:.1%}'.format(0.25)       # '25.0%'
'{0}{1}'.format('a', 'b')   # 'ab'
'{1}{0}'.format('a', 'b')   # 'ba'
'{num:}'.format(num=7)      # '7' (named args)
```

## List (mutable, indexed, ordered container)

Indexed from zero to length-1

```
a = [] # the empty list
a = ['dog', 'cat', 'bird'] # simple list
a = [[1, 2], ['a', 'b']] # nested lists
a = [1, 2, 3] + [4, 5, 6] # concatenation
a = [1, 2, 3] * 456 # replication
a = list(x) # conversion
```

## List comprehensions (can be nested)

Comprehensions: a tight way of creating lists

```
t3 = [x*3 for x in [5, 6, 7]] # [15, 18, 21]
z = [complex(x, y) for x in range(0, 4, 1)
     for y in range(4, 0, -1) if x > y]
# z --> [(2+1j), (3+2j), (3+1j)]
```

## Iterating lists

```
L = ['dog', 'cat', 'turtle']
for item in L:
    print (item)
for index, item in enumerate(L):
    print (index, item)
```

## Searching lists

```
L = ['dog', 'cat', 'turtle']; value = 'cat'
if value in L:
    count = L.count(value)
    first_occurrence = L.index(value)
if value not in L:
    print 'list is missing {}'.format(value)
```

## List methods (not a complete list)

Method	What it does
<b>l.append(x)</b>	Add x to end of list
<b>l.extend(other)</b>	Append items from other
<b>l.insert(pos, x)</b>	Insert x at position
<b>del l[pos]</b>	Delete item at pos
<b>l.remove(x)</b>	Remove first occurrence of x; An error if no x
<b>l.pop([pos])</b>	Remove last item from list (or item from pos); An error if empty list
<b>l.index(x)</b>	Get index of first occurrence of x; An error if x not found
<b>l.count(x)</b>	Count the number of times x is found in the list
<b>l.sort()</b>	In place list sort
<b>l.reverse(x)</b>	In place list reversal

## List slicing

```
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 2nd
```

**Note:** All Python sequence types support the above index slicing (strings, lists, tuples, bytearray, buffers, and xrange objects)

## Set (unique, unordered container)

A Python set is an unordered, mutable collection of unique hashable objects.

```
a = set() # empty set
a = {'red', 'white', 'blue'} # simple set
a = set(x) # convert list
```

**Trap:** {} creates empty dict, not an empty set

## Set comprehensions

```
# a set of selected letters...
s = {e for e in 'ABCHJADC' if e not in 'AB'}
# --> {'H', 'C', 'J', 'D'}
# a set of tuples ...
s = {(x,y) for x in range(-1,2)
      for y in range(-1,2)}
```

**Trap:** set contents need to be immutable to be hashable. So you can have a set of tuples, but not a set of lists.

## Iterating a set

```
for item in set:
    print (item)
```

## Searching a set

```
if item in set:
    print (item)
if item not in set:
    print ('{} is missing'.format(item))
```

## Set methods (not a complete list)

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

## Frozenset

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

```
f = frozenset(s) # convert set
f = frozenset(o) # convert other
```

## Dictionary (indexed, unordered map-container)

A mutable hash map of unique key=value pairs.

```
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 = 'a=apple,b=bird,c=cat,d=dog,e=egg'
a = dict(i.split("=") for i in s.split(","))
# {'a': 'apple', 'c': 'cat', 'b': 'bird',
# 'e': 'egg', 'd': 'dog'}
```

## Dictionary comprehensions

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

```
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

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

## Searching a dictionary

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

## Merging two dictionaries

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

## Dictionary methods (not a complete list)

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



## Key functions (not a complete list)

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

## Using functions

When called, functions can take positional and named arguments.

For example:

```
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

```
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.

```
def nasty(value=[]): # <-- mutable arg  
    value.append('a')  
    return value  
print (nasty ()) # --> ['a']  
print (nasty ()) # --> ['a', 'a']  
  
def better(val=None):  
    val = [] if val is None else val  
    value.append('a')  
    return value
```

## Lambda (inline expression) functions:

```
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().

```
# 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.

```
def derivative(f, dx):  
    """Return a function that approximates  
    the derivative of f using an interval  
    of dx, which should be appropriately  
    small.  
    """  
    def _function(x):  
        return (f(x + dx) - f(x)) / dx  
    return _function #from derivative(f, dx)  
  
f_dash_x = derivative(lambda x: x*x, 0.00001)  
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 \
            '(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())
print (Point.count) # check static variable
Point.static_eg(9) # check static method
```

## 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.

Magic method	What it does
<code>__init__(self,[...])</code>	Constructor
<code>__del__(self)</code>	Destructor pre-garbage collection
<code>__str__(self)</code>	Human readable string for class contents. Called by <code>str(self)</code>
<code>__repr__(self)</code>	Machine readable unambiguous Python string expression for class contents. Called by <code>repr(self)</code> Note: <code>str(self)</code> will call <code>__repr__</code> if <code>__str__</code> is not defined.
<code>__eq__(self, other)</code>	Behaviour for <code>==</code>
<code>__ne__(self, other)</code>	Behaviour for <code>!=</code>
<code>__lt__(self, other)</code>	Behaviour for <code>&lt;</code>
<code>__gt__(self, other)</code>	Behaviour for <code>&gt;</code>
<code>__le__(self, other)</code>	Behaviour for <code>&lt;=</code>
<code>__ge__(self, other)</code>	Behaviour for <code>&gt;=</code>
<code>__add__(self, other)</code>	Behaviour for <code>+</code>
<code>__sub__(self, other)</code>	Behaviour for <code>-</code>
<code>__mul__(self, other)</code>	Behaviour for <code>*</code>
<code>__div__(self, other)</code>	Behaviour for <code>/</code>
<code>__mod__(self, other)</code>	Behaviour for <code>%</code>
<code>__pow__(self, other)</code>	Behaviour for <code>**</code>
<code>__pos__(self, other)</code>	Behaviour for unary <code>+</code>
<code>__neg__(self, other)</code>	Behaviour for unary <code>-</code>
<code>__hash__(self)</code>	Returns an int when <code>hash()</code> called. Allows class instance to be put in a dictionary
<code>__len__(self)</code>	Length of container
<code>__contains__(self, i)</code>	Behaviour for <code>in</code> and <code>not in</code> operators
<code>__missing__(self, i)</code>	What to do when dict key <code>i</code> is missing
<code>__copy__(self)</code>	Shallow copy constructor
<code>__deepcopy__(self, memodict={})</code>	Deep copy constructor
<code>__iter__(self)</code>	Provide an iterator
<code>__nonzero__(self)</code>	Called by <code>bool(self)</code>
<code>__index__(self)</code>	Called by <code>x[self]</code>
<code>__setattr__(self, name, val)</code>	Called by <code>self.name = val</code>
<code>__getattr__(self, name)</code>	Called by <code>self.name</code>
<code>__getattr__(self, name)</code>	Called when <code>self.name</code> does not exist
<code>__delattr__(self, name)</code>	Called by <code>del self.name</code>
<code>__getitem__(self, key)</code>	Called by <code>self[key]</code>
<code>__setitem__(self, key, val)</code>	Called by <code>self[key] = val</code>
<code>__delitem__(self, key)</code>	<code>del self[key]</code>