PyCon 2015 - Python Epiphanies

Overview

This tutorial, presented at PyCon 2015 in Montreal by Stuart Williams, is intended for intermediate Python users looking for a deeper understanding of the language. It attempts to correct some common misperceptions of how Python works. Python is very similar to other programming languages, but quite different in some subtle but important ways.

You'll learn by seeing and doing. We'll mostly use the interactive Python interpreter prompt, aka the Read Eval Print Loop (REPL). I'll be using Python version 3.4 but most of this will work identically in earlier versions.

Most exercise sections start out simple but increase quickly in difficulty in order to give more advanced students a challenge, so don't expect to finish all the exercises in each section. I encourage you to revisit them later.

I encourage you to *not* copy and paste code from this document when you do the exercises. By typing the code you will learn more. Also pause before you hit the Enter key and try to predict what will happen.

Python Epiphanies Tutorial
Presented at PyCon 2015
April 8th, 2015
Montreal, Quebec, Canada

Stuart Williams
stuart@swilliams.ca
@ceilous

Objects

```
>>> # Create objects via literals                                    0
>>> 1                                                               1
>>> 3.14                                                            2
>>> 'walk'                                                          3
>>> # Pre-created objects via constants                             4
>>> True                                                            5
>>> False                                                           6
```

Everything in Python (at runtime) is an object and has:

- a single *value*,
- a single *type*,
- some number of *attributes*,
- one or more *base classes*,
- a single *id*, and
- (zero or) one or more *names* (in one or more namespaces).

```
>>> # Object have types                                            7
>>> type(1)                                                        8
>>> type(3.14)                                                     9
>>> type('walk')                                                  10
>>> type(True)                                                    11

>>> # Objects have attributes                                      12

>>> True.__doc__                                                  13
>>> 'walk'.__add__                                                14
>>> callable('walk'.__add__)                                      15
>>> 'walk'.__add__('about')                                       16
>>> (2.0).hex                                                     17
```

```
>>> (2.0).hex()
>>> (4.0).hex()

>>> # Objects have base classes
>>> import inspect
>>> inspect.getmro(3)
>>> inspect.getmro(type(3))
>>> inspect.getmro(type('walk'))
>>> inspect.getmro(type(True))

>>> # Base classes are stored in attributes
>>> True.__class__
>>> True.__class__.__bases__
>>> True.__class__.__bases__[0]
>>> True.__class__.__bases__[0].__bases__[0]
>>> inspect.getmro(type(True))

>>> # Every object has one id (memory address in CPython)
>>> id(3)
>>> id(3.14)
>>> id('walk')
>>> id(True)

>>> # Create objects by calling an object (function, method, class)
>>> abs
>>> callable(abs)
>>> abs(-3)
>>> int
>>> callable(int)
>>> int(3.14)
>>> 'walk'.__len__
>>> 'walk'.__len__()
>>> 'walk'.__add__
>>> 'walk'.__add__('about')
>>> dict
>>> dict()
>>> dict(pi=3.14, e=2.71)
```

Exercises: Objects

```
>>> 5.0
>>> dir(5.0)
>>> 5.0.__add__
>>> callable(5.0.__add__)
>>> 5.0.__add__()
>>> 5.0.__add__(4)

>>> 4.__add__
>>> (4).__add__
>>> (4).__add__(5)

>>> import sys
>>> size = sys.getsizeof
>>> size('w')
>>> size('walk')
>>> size(2)
>>> size(2**30 - 1)
>>> size(2**30)
>>> size(2**60-1)
>>> size(2**60)
>>> size(2**1000)
```

Names

```
>>> # We can add names to refer to objects

>>> # Adding names to a namespace is like updating a dictionary
>>> dir()
>>> def __names():
```

```
...     return dict([(k, v) for (k, v) in globals().items()
...     if not k.startswith('__')])

>>> __names()                                                                 74
>>> a                                                                         75
>>> a = 300                                                                   76
>>> __names()                                                                 77
>>> a                                                                         78
>>> a = 400                                                                   79
>>> __names()                                                                 80
>>> a                                                                         81
>>> b = a                                                                     82
>>> b                                                                         83
>>> a                                                                         84
>>> __names()                                                                 85
>>> id(a)                                                                     86
>>> id(b)                                                                     87
>>> a is b                                                                    88
>>> a = 'walk'                                                                89
>>> a                                                                         90
>>> b                                                                         91
>>> del a                                                                     92
>>> __names()                                                                 93
>>> del b                                                                     94

>>> # object attributes are like dictionaries of dictionaries                 95

>>> my_namespace = {}                                                         96
>>> my_namespace['r'] = {}                                                    97
>>> my_namespace['r']['x'] = 1.0                                              98
>>> my_namespace['r']['y'] = 2.0                                              99
>>> my_namespace['r']['x']                                                   100
>>> my_namespace['r']                                                        101
>>> my_namespace                                                             102

>>> # For Python < 3.3 use class SimpleNamespace: pass                       103
>>> import types                                                             104
>>> r = types.SimpleNamespace()                                              105

>>> r.x = 1.0                                                                106
>>> r.y = 1.0                                                                107
>>> r.x                                                                      108
>>> r.y                                                                      109

>>> # 'is' checks identity (via 'id'), not equality                          110
>>> i = 10                                                                   111
>>> j = 10                                                                   112
>>> i is j                                                                   113

>>> i = 500                                                                  114
>>> j = 500                                                                  115
>>> i is j                                                                   116

>>> # CPython-specific optimizations                                         117
>>> id(254)                                                                  118
>>> id(255)                                                                  119
>>> id(256)                                                                  120
>>> id(257)                                                                  121
>>> id(258)                                                                  122
```

Exercises: Names

Restart Python to unclutter the local namespace.

```
>>> i                                                                        123
>>> dir()                                                                    124
>>> i = 1                                                                    125
>>> i                                                                        126
>>> dir()                                                                    127
>>> type(i)                                                                  128
>>> j = i                                                                    129
>>> i is j                                                                   130
```

```
>>> m = [1, 2, 3]
>>> m
>>> n = m
>>> n
>>> dir()
>>> m is n
>>> m[1] = 'two'
>>> m
>>> n

>>> s = t = 'hello'
>>> s
>>> s is t, id(s), id(t)
>>> s += ' there'
>>> s
>>> s is t, id(s), id(t)

>>> m = n = [1, 2, 3]
>>> m
>>> m is n, id(m), id(n)
>>> m += [4]
>>> m
>>> n
>>> m is n, id(m), id(n)

>>> int.__add__
>>> int.__add__ = int.__sub__

>>> new_object = object()
>>> dir(None)
>>> len(dir(None)), len(dir(new_object))
>>> set(dir(None)) - set(dir(new_object))

>>> import sys
>>> refs = sys.getrefcount
>>> refs(None)
>>> refs(object)
>>> refs(new_object)
>>> sentinel = object()
>>> sentinel == object()
>>> sentinel == sentinel

>>> refs(1)
>>> refs(2)
>>> refs(25)
>>> [sys.getrefcount(i) for i in range(266)]
```

Namespaces

A *namespace* is a mapping from valid identifier names to objects. Think of it as a dictionary.

Assignment is a namespace operation, not an operation on variables or objects.

A *scope* is a section of Python code where a namespace is *directly* accessible.

For a *directly* accessible namespace you access values in the (namespace) dictionary by key alone, e.g. `s2` instead of `my_namespace['s2']`.

For *indirectly* accessible namespace you access values via dot notation, e.g. `dict.__doc__` or `sys.version_info.major`.

The (*direct*) namespace search order is (from http://docs.python.org/tutorial):

- #1: the innermost scope contains local names
- #2: the namespaces of enclosing functions, searched starting with the nearest enclosing scope; (or the module if outside any function)
- #3: the middle scope contains the current module's global names
- #4: the outermost scope is the namespace containing built-in names

All namespace *changes* happen in the local scope (i.e. in the current scope in which the namespace-changing code executes):

- `=` i.e. assignment
- `del`
- `import`
- `def`
- `class`

Other namespace changes:

- function parameters: `def foo(NEW_NAME):`
- for loop: `for NEW_NAME in ...`
- except clause: `Exception as NEW_NAME:`
- with clause: `with open(filename) as NEW_NAME:`
- docstrings: `__doc__`

```
>>> len                                                          171
>>> def f1():                                                    172
...     def len():
...         len = range(3)
...         print("In f1's len(), len = {}".format(len))
...         return 'Returning len: {!r}'.format(len)
...     print('In f1(), len = {}'.format(len))
...     return len()
```

```
>>> f1()                                                         173
>>> def f2():                                                    174
...     def len():
...         # len = range(3)
...         print("In f2's len(), len = {}".format(len))
...         return 'Returning len: {!r}'.format(len)
...     print('In f2(), len = {}'.format(len))
...     return len()
```

```
>>> f2()                                                         175
>>> len                                                          176
>>> len = 99                                                     177
>>> def f3(s):                                                   178
...     print('len(s) == {}'.format(len(s)))
```

```
>>> f3('walk')                                                   179
>>> len                                                          180
>>> del len                                                      181
>>> len                                                          182
>>> f3('walk')                                                   183
>>> pass                                                         184
>>> pass = 3                                                     185
>>> del                                                          186
```

Keywords:

```
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
```

Namespaces: function locals

Let's look at some surprising behaviour:

```
>>> x = 1                                                        187
>>> def test1():                                                 188
...     print('In test1 x ==', x)
```

```
>>> test1()                                                      189
```

```
>>> def test2():                                                 190
...     x = 2
```

```
...     print('In test2 x ==', x)

>>> x                                                                    191
>>> test2()                                                              192
>>> x                                                                    193

>>> def test3():                                                         194
...     print('In test3  ==', x)
...     x = 3

>>> x                                                                    195
>>> test3()                                                              196
>>> x                                                                    197

>>> test3.__code__                                                       198
>>> test3.__code__.co_argcount                                           199
>>> test3.__code__.co_name                                               200
>>> test3.__code__.co_names                                              201
>>> test3.__code__.co_nlocals                                            202
>>> test3.__code__.co_varnames  # tuple of local names                   203
```

"If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged)." [Python tutorial section 9.2 at http://docs.python.org/tutorial]

```
>>> def test4():                                                         204
...     global x
...     print('In test4 before, x ==', x)
...     x = 4
...     print('In test4 after, x ==', x)

>>> x                                                                    205
>>> test4()                                                              206
>>> x                                                                    207

>>> test4.__code__.co_varnames                                           208

>>> def test5():                                                         209
...     x = 5
...     def test6():
...         nonlocal x
...         print('test6 before x ==', x)
...         x = 6
...         print('test6 after x ==', x)
...     print('test5 before x ==', x)
...     test6()
...     print('test5 after x ==', x)

>>> x = 1                                                                210
>>> x                                                                    211
>>> test5()                                                              212
>>> x                                                                    213
```

The Local Namespace

```
>>> help(dir)                                                            214
>>> dir()                                                                215

>>> import builtins, collections, inspect, textwrap                      216

>>> fill = textwrap.TextWrapper(width=60).fill                           217
>>> def pfill(pairs):                                                    218
...     print(fill(' '.join(
...         (n for (n, o) in sorted(pairs)))))

>>> members = set([                                                      219
...     m for m in inspect.getmembers(builtins)
...     if not m[0].startswith('_')])

>>> len(members)                                                         220
>>> exceptions = [                                                       221
```

```
...     (name, obj) for (name, obj) in members
...     if inspect.isclass(obj) and
...     issubclass(obj, BaseException)]

>>> members -= set(exceptions)

>>> len(exceptions)
>>> pfill(exceptions)

>>> len(members)
>>> pfill(members)

>>> type(int)
>>> type(len)

>>> bnames = collections.defaultdict(set)
>>> for name, obj in members:
...     bnames[type(obj)].add((name, obj))

>>> for typ in [type(int), type(len)]:
...     pairs = bnames.pop(typ)
...     print(typ)
...     pfill(pairs)
...     print()

>>> for typ, pairs in bnames.items():
...     print('{}: {}'.format(typ, ' '.join((n for (n, o) in pairs))))
```

Exercises: Namespaces

```
>>> locals().keys()
>>> globals().keys()
>>> locals() == globals()
>>> locals() is globals()  # Not always True

>>> x
>>> locals()['x']
>>> locals()['x'] = 1
>>> locals()['x']
>>> x
>>> dir()
```

Most builtins are unsurprising cases of type `exception`, type `built-in function`, or `type`. Explore some of the following suprising ones via introspection (e.g. `type`, `inspect.getmro`, and `help`) or the Python documentation:

- `...`
- `Ellipsis`
- `NotImplementedType`
- `True`, `None`

```
>>> import inspect
>>> inspect.getmro(type(True))
```

Namespace Changes

Remember, these change or modify a namespace:

- assignment
- [`globals()` and `locals()`]
- `import`
- `def`
- `class`
- `del`
- [also `def`, `for`, `except`, `with`, docstrings]

Next we'll explore `import`.

```
>>> dir()
>>> import pprint
>>> dir()
>>> pprint
>>> dir(pprint)

>>> print('\n'.join([a for a in dir(pprint)
...                   if not a.startswith('_')]))

>>> pprint.pformat
>>> pprint.pprint
>>> pprint.foo
>>> foo
>>> pprint.foo = 'Python is dangerous'
>>> pprint.foo

>>> from pprint import pformat as pprint_pformat
>>> dir()
>>> pprint.pformat is pprint_pformat
>>> pprint
>>> pprint.pformat

>>> del pprint
>>> import pprint as pprint_module
>>> dir()
>>> pprint_module.pformat is pprint_pformat

>>> module_name = 'string'
>>> import importlib
>>> string_module = importlib.import_module(module_name)
>>> string_module.ascii_uppercase
>>> string
>>> import module_name
>>> import 'string'
>>> import string
```

File structure:

```
folder1/
  file1.py

module1/
  __init__.py -- zero length
  file1.py:
    attribute1 = 1
```

```
>>> dir()
>>> import folder1
>>> dir(folder1)
>>> hasattr(folder1, '__path__')
>>> import folder1.file1
>>> dir(folder1.file1)
>>> import module1
>>> dir()
>>> dir(module1)
>>> import module1.file1
>>> dir()
>>> dir(module1)
>>> dir(module1.file1)
>>> from module1 import file1
>>> dir()
>>> dir(file1)
```

Exercises: The import statement

```
>>> import pprint
>>> dir(pprint)
>>> pprint.__doc__
>>> pprint.__file__
>>> pprint.__name__
>>> pprint.__package__
```

```
>>> from pprint import *
>>> dir()

>>> import importlib
>>> importlib.reload(csv)
>>> importlib.reload('csv')
>>> import csv
>>> importlib.reload('csv')
>>> importlib.reload(csv)

>>> import sys
>>> sys.path
```

Functions

```
>>> def f():
...     pass

>>> f.__name__
>>> dir()
>>> f.__name__ = 'g'
>>> dir()
>>> f.__name__
>>> f
>>> f.__qualname__  # Python >= 3.3
>>> f.__qualname__ = 'g'
>>> f
>>> f.__dict__
>>> f.foo = 'bar'
>>> f.__dict__

>>> def f(a, b, k1='k1', k2='k2',
...        *args, **kwargs):
...     print('a: {!r}, b: {!r}, '
...         'k1: {!r}, k2: {!r}'
...         .format(a, b, k1, k2))
...     print('args:', repr(args))
...     print('kwargs:', repr(kwargs))

>>> f.__defaults__
>>> f(1, 2)
>>> f(a=1, b=2)
>>> f(b=1, a=2)
>>> f(1, 2, 3)
>>> f(1, 2, k2=4)
>>> f(1, k1=3)
>>> f(1, 2, 3, 4, 5, 6)
>>> f(1, 2, 3, 4, keya=7, keyb=8)
>>> f(1, 2, 3, 4, 5, 6, keya=7, keyb=8)

>>> def g(a, b, *args, c=None):
...     print('a: {!r}, b: {!r}, '
...         'args: {!r}, c: {!r}'
...         .format(a, b, args, c))

>>> g.__defaults__
>>> g.__kwdefaults__
>>> g(1, 2, 3, 4)
>>> g(1, 2, 3, 4, c=True)

>>> def h(a=None, *args, b=None):
...     print('a: {!r}, args: {!r}, '
...         'b: {!r}'
...         .format(a, args, b))

>>> h.__defaults__
>>> h.__kwdefaults__
>>> h(1, 2, 3, 4)
>>> h(1, 2, 3, 4, b=True)
```

Exercises: Functions

```
>>> def f(*args, **kwargs):
...     print(repr(args), repr(kwargs))

>>> f(1)
>>> f(1, 2)
>>> f(1, a=3, b=4)

>>> t = 1, 2
>>> t
>>> d = dict(k1=3, k2=4)
>>> d
>>> f(*t)
>>> f(**d)
>>> f(*t, **d)

>>> m = 'one two'.split()
>>> f(1, 2, *m)

>>> locals()
>>> name = 'Dad'
>>> 'Hi {name}'.format(**locals())

>>> def f2(a: 'x', b: 5, c: None, d:list) -> float:
...     pass

>>> f2.__annotations__
```

Lists are mutable, strings are not

```
>>> # First with ``=`` and ``+``, then with ``+=``:

>>> s1 = s2 = 'hello'
>>> s1 = s1 + ' there'
>>> s1, s2

>>> s1 = s2 = 'hello'
>>> s1 += ' there'
>>> s1, s2

>>> m1 = m2 = [1, 2, 3]
>>> m1 = m1 + [4]
>>> m1, m2

>>> m1 = m2 = [1, 2, 3]
>>> m1 += [4]
>>> m1, m2

>>> # Why?
>>> # += is its own operator, not identical to foo = foo + 1

>>> import codeop, dis
>>> dis.dis(codeop.compile_command('m = [1, 2, 3]; m += [4]'))
>>> dis.dis(codeop.compile_command("s = 'hello'; s += ' there'"))

>>> m = [1, 2, 3]
>>> m
>>> m.__iadd__([4])  # note return value
>>> m

>>> s1.__iadd__(' there')
```

The difference is because `str.__iadd__ copies`, but `list.__iadd__` mutates.

https://docs.python.org/3/reference/datamodel.html#object.__iadd__:

These methods are called to implement the augmented arithmetic assignments (+=, etc.). These methods should attempt to do the operation in-place (modifying self) and return the result (which could be, but does not have to be, self). If a specific method is not defined, the augmented assignment falls back to the normal methods.

```
>>> t1 = (1, 2)
>>> t1[0] += 1
>>> t2[0] = 1 + 1
```

```
>>> t2 = (['one'],)
>>> t2
>>> t2[0] += ['two']
>>> t2
```

```
>>> t2 = (['one'],)
>>> t2
>>> result = t2[0].__iadd__(['two'])
>>> result
>>> t2[0]
>>> t2[0] = result
>>> t2
```

Parameters by reference

```
>>> def test1(s):
...     print('Before:', s)
...     s += ' there'
...     print('After:', s)
```

```
>>> str2 = 'hello'
>>> str2
>>> test1(str2)
>>> str2
>>> test1('hello')
```

```
>>> def test2(m):
...     print('Before:', m)
...     m += [4]
...     print('After:', m)
```

```
>>> list3 = [1, 2, 3]
>>> list3
>>> test2(list3)
>>> list3
```

Decorators

A decorator modifies an existing function:

- Before it starts executing
  - Including changing parameters
- After it's done executing
  - Including changing what is returned

```
>>> def square(n):
...     return n * n
```

```
>>> square(2)
>>> square(3)
```

```
>>> def trace_function(f):
...     def new_f(*args):
...         print(
...             'called {}({!r})'
...             .format(f.__qualname__, *args))
...         result = f(*args)
...         print('returning', result)
...         return result
...     return new_f
```

```
>>> traced_square = trace_function(square)
>>> traced_square(2)
>>> traced_square(3)
```

```
>>> @trace_function
>>> def cube(n):
...     return n ** 3

>>> cube(2)
>>> cube(3)

>>> def memoize(f):
...     cache = {}
...     def memoized_f(*args):
...         if args in cache:
...             print('Hit!')
...             return cache[args]
...         if args not in cache:
...             result = f(*args)
...             cache[args] = result
...             return result
...     return memoized_f

>>> @memoize
>>> def cube(n):
...     return n ** 3

>>> cube(2)
>>> cube(3)
>>> cube(2)
```

Exercises: Decorators

A decorator is a function that takes a function as a parameter and *typically* returns a new function, but it can return anything. The following code misuses decorators to make you think about their mechanics, which are really quite simple. What does it do?

```
>>> del x
>>> x

>>> def return_3(f):
...     return 3

>>> @return_3
>>> def x():
...     pass

>>> x
>>> type(x)
```

Here's equivalent code without using `@decorator` syntax:

```
>>> # Without decorator syntax
>>> del x
>>> x
>>> def x():
...     pass

>>> x
>>> x = return_3(x)
>>> x
```

Another decorator:

```
>>> def doubler(f):
...     def new_f(*args):
...         return 2 * f(*args)
...     return new_f

>>> @doubler
>>> def cube(n):
...     return n ** 3

>>> cube(1)
>>> cube(2)
```

```
>>> @doubler                                                                    442
>>> def create_list(a, b):                                                      443
...     return [a, b]

>>> create_list(1, 2)                                                           444

>>> class Counter:                                                              445
...     def __init__(self):
...         self.count = 0
...     def __call__(self, *args):
...         self.count += 1

>>> c = Counter()                                                               446
>>> c.count                                                                     447
>>> c()                                                                         448
>>> c()                                                                         449
>>> c.count                                                                     450

>>> class function_counter:                                                     451
...     def __init__(self, f):
...         print('function_counter.__init__ called')
...         self.f = f
...         self.count = 0
...     def __call__(self, *args):
...         print('function_counter.__call__ called')
...         self.count += 1
...         return self.f(*args)

>>> def plural(s):                                                              452
...     return s + 's'

>>> plural_counter = function_counter(plural)                                   453

>>> plural_counter('dog')                                                       454
>>> plural_counter('cat')                                                       455
>>> plural_counter.count                                                        456

>>> @function_counter                                                           457
>>> def plural(s):                                                              458
...     return s + 's'

>>> plural('dog')                                                               459
>>> plural.count                                                                460
```

The class statement

1. The `class` statement, which starts a block of code, creates a new namespace and all the name changes in the block, i.e. assignment and function definitions, are made in that new namespace. It also creates a name for the class in the namespace of the module where it appears.

2. Instances of a class are created by calling the class: `ClassName()` or `ClassName(parameters)`.

`ClassName.__init__(<new object>, ...)` is called automatically, passing as first parameter an object, the new instance of the class, which was created by a call to `__new__()`.

3. Accessing an attribute `method_name` on a class instance returns a *method object*, if `method_name` references a method (in `ClassName` or its superclasses). A method object binds the class instance as the first parameter to the method.

Number class:

```
class Number():
  def __init__(self, amount):
    self.amount = amount

  def add(self, value):
    print('Call: add({!r}, {})'
          .format(self, value))
    return self.amount + value
```

```
>>> class Number():                                                             461
```

```
...     """A number class."""
...     # This comment satisfies the REPL
...     __version__ = '1.0'
...     #
...     def __init__(self, amount):
...         self.amount = amount
...     #
...     def add(self, value):
...         """Add a value to the number."""
...         print('Call: add({!r}, {})'.format(self, value))
...         return self.amount + value

>>> Number                                                                    462
>>> Number.__version__                                                        463
>>> Number.__doc__                                                            464
>>> help(Number)                                                              465
>>> Number.__init__                                                           466
>>> Number.add                                                                467
>>> dir(Number)                                                               468
>>> def dirp(obj):                                                            469
...     return [n for n in dir(obj) if not n.startswith('__')]

>>> dirp(Number)                                                              470

>>> number2 = Number(2)                                                       471
>>> number2.amount                                                            472
>>> number2                                                                   473
>>> number2.__init__                                                          474
>>> number2.add                                                               475
>>> dirp(number2)                                                             476
>>> set(dir(number2)) - set(dir(Number))                                      477
>>> set(dir(Number)) - set(dir(number2))                                      478
>>> number2.__dict__                                                          479
>>> Number.__dict__                                                           480

>>> number2.add                                                               481
>>> number2.add(3)                                                            482
>>> Number.add                                                                483
>>> # Warning - unusual code ahead                                            484
>>> Number.add(2)                                                             485
>>> Number.add(2, 3)                                                          486
>>> Number.add(number2, 3)                                                    487

>>> number2.add(3)                                                            488

>>> # Warning - weird code ahead                                              489
>>> def set_double_amount(number, amount):                                    490
...     number.amount = 2 * amount

>>> Number.__init__                                                           491
>>> help(Number.__init__)                                                     492
>>> Number.__init__ = set_double_amount                                       493
>>> Number.__init__                                                           494
>>> help(Number.__init__)                                                     495

>>> number4 = Number(2)                                                       496
>>> number4.amount                                                            497
>>> number4.add(5)                                                            498
>>> number4.__init__                                                          499
>>> number2.__init__                                                          500

>>> def multiply_by(number, value):                                           501
...     return number.amount * value

>>> # I intentionally make a mistake...                                       502
>>> number4.mul = multiply_by                                                 503
>>> number4.mul                                                               504
>>> number4.mul(5)                                                            505
>>> number4.mul(number4, 5)                                                   506
>>> # Where's the mistake?                                                    507
>>> number10 = Number(5)                                                      508
>>> number10.mul                                                              509
>>> dirp(number10)                                                            510
>>> dirp(Number)                                                              511
>>> dirp(number4)                                                             512
```

```
>>> Number.mul = multiply_by
>>> number10.mul(5)
>>> number4.mul(5)
>>> dirp(number4)
>>> number4.__dict__
>>> number4.mul
>>> del number4.mul
>>> dirp(number4)
>>> number4.mul
>>> Number.mul
>>> number4.mul(5)

>>> # Behind the curtain
>>> Number
>>> number4
>>> Number.add
>>> number4.add
>>> dirp(number4.add)
>>> set(dir(number4.add)) - set(dir(Number.add))

>>> number4.add.__self__
>>> number4.add.__self__ is number4
>>> add_value_to_number_4 = number4.add
>>> add_value_to_number_4(6)

>>> number4.add.__func__
>>> number4.add.__self__

>>> number4.add.__func__ is Number.add
>>> number4.add.__func__ is number10.add.__func__

>>> number4.add(5)
>>> number4.add.__func__(number4.add.__self__, 5)
```

The type function for classes

"The class statement is just a way to call a function, take the result, and put it into a namespace." -- Glyph Lefkowitz in *Turtles All The Way Down...* at PyCon 2010

`type(name, bases, dict)` is the function that gets called when a `class` statement is used to create a class.

```
>>> print(type.__doc__)

>>> # Let's use the type function to build a class:
>>> def _init(self, amount):
...     self.amount = amount

>>> def _add(self, value):
...     return self.amount + value

>>> Number = type(
...     'Number',
...     (object,),
...     { '__init__': _init,
...       'add': _add,
...     })

>>> number3 = Number(3)
>>> type(number3)
>>> number3.__class__
>>> number3.__dict__
>>> number3.amount
>>> number3.add(4)

>>> # The *right* way:
>>> class Number:
...     def __init__(self, amount):
...         self.amount = amount
...     #
...     def add(self, value):
...         return self.amount + value
```

```
>>> number2 = Number(2)
>>> number2.amount
>>> number2.add(3)
```

By default, classes are constructed using type(). The class body is executed in a new namespace and the class name is bound locally to the result of type(name, bases, namespace).

The class creation process can be customised by passing the metaclass keyword argument in the class definition line, or by inheriting from an existing class that included such an argument.

https://docs.python.org/3.4/reference/datamodel.html#customizing-class-creation

```
>>> class Number(metaclass=type):  # default metaclass is type
...     def __init__(self, amount):
...         self.amount = amount
```

Exercises: The class statement

What does the following code do? Note that `return_5` ignores its arguments.

```
>>> def return_5(name, bases, namespace):
...     return 5
...
... return_5(None, None, None)

>>> x = return_5(None, None, None)
>>> x
>>> type(x)
>>> dir()

>>> class y(metaclass=return_5):
...     pass

>>> dir()
>>> y
>>> type(y)
```

We saw how decorators are applied to functions. They can also be applied to classes. What does the following code do?

```
>>> # Apply a decorator to a class
>>> def return_6(klass):
...     return 6

>>> return_6(None)

>>> dir()
>>> @return_6
>>> class z:
...     pass

>>> dir()
>>> z
>>> type(z)
```

Class decorator example

```
>>> def class_counter(klass):
...     """Modify klass to count class instantiations"""
...     klass.count = 0
...     klass.__init_orig__ = klass.__init__
...     def new_init(self, *args, **kwargs):
...         klass.count += 1
...         klass.__init_orig__(self, *args, **kwargs)
...     klass.__init__ = new_init
...     return klass

>>> @class_counter
>>> class TC:
```

```
...     pass
...
... TC.count
... TC()
... TC()
... TC.count
```

Standard class methods

- __new__, __init__, __del__, __repr__, __str__, __format__
- __getattr__, __getattribute__, __setattr__, __delattr__, __call__, __dir__
- __len__, __getitem__, __missing__, __setitem__, __delitem__, __contains__, __iter__, __next__
- __lt__, __le__, __gt__, __ge__, __eq__, __ne__, __cmp__, __nonzero__, __hash__
- __add__, __sub__, __mul__, __div__, __floordiv__, __mod__, __divmod__, __pow__, __and__, __xor__, __or__, __lshift__, __rshift__, __neg__, __pos__, __abs__, __invert__, __iadd__, __isub__, __imul__, __idiv__, __itruediv__, __ifloordiv__, __imod__, __ipow__, __iand__, __ixor__, __ior__, __ilshift__, __irshift__
- __int__, __long__, __float__, __complex__, __oct__, __hex__, __coerce__
- __radd__, __rsub__, __rmul__, __rdiv__, etc.
- __enter__, __exit__

```
>>> class UpperAttr:                                                    579
...     """
...     A class that returns uppercase values
...     on uppercase attribute access.
...     """
...     def __getattr__(self, name):
...         if name.isupper():
...             if name.lower() in self.__dict__:
...                 return self.__dict__[
...                     name.lower()].upper()
...         raise AttributeError(
...             "'{}' object has no attribute {}."
...             .format(self, name))
```

```
>>> d = UpperAttr()                                                    580
>>> d.__dict__                                                         581
>>> d.foo = 'bar'                                                      582
>>> d.foo                                                              583
>>> d.__dict__                                                         584
>>> d.FOO                                                              585
>>> d.baz                                                              586
```

Optional Exercises: Standard class methods

Try the following (in a file if that's easier):

```
>>> class Get:                                                         587
...     def __getitem__(self, key):
...         print('called __getitem__({} {})'
...             .format(type(key), repr(key)))
```

```
>>> g = Get()                                                          588
>>> g[1]                                                               589
>>> g[-1]                                                              590
>>> g[0:3]                                                             591
>>> g[0:10:2]                                                          592
>>> g['Jan']                                                           593
>>> g[g]                                                               594
```

```
>>> m = list('abcdefghij')                                             595
>>> m[0]                                                               596
>>> m[-1]                                                              597
>>> m[::2]                                                             598
>>> s = slice(3)                                                       599
>>> m[s]                                                               600
>>> m[slice(1, 3)]                                                     601
>>> m[slice(0, 2)]                                                     602
```

```
>>> m[slice(0, len(m), 2)]
>>> m[::2]
```

Properties

```
>>> class PropertyExample:
...     def __init__(self):
...         self._x = None
...     def getx(self):
...         print('called getx()')
...         return self._x
...     def setx(self, value):
...         print('called setx()')
...         self._x = value
...     def delx(self):
...         print('del x')
...         del self._x
...     x = property(getx, setx, delx, "The 'x' property.")

>>> p = PropertyExample()

>>> p.setx('foo')
>>> p.getx()
>>> p.x = 'bar'
>>> p.x
>>> del p.x
```

Iterators

- A `for` loop evaluates an expression to get an *iterable* and then calls `iter()` to get an iterator.
- The iterator's `__next__()` method is called repeatedly until `StopIteration` is raised.
- `iter(foo)`
    - checks for `foo.__iter__()` and calls it if it exists
    - else checks for `foo.__getitem__()` and returns an object which calls it starting at zero and handles `IndexError` by raising `StopIteration`.

```
>>> class MyList:
...     def __init__(self, sequence):
...         self.items = sequence
...     #
...     def __getitem__(self, key):
...         print('called __getitem__({})'
...               .format(key))
...         return self.items[key]

>>> m = MyList(['a', 'b', 'c'])

>>> m.__getitem__(0)
>>> m.__getitem__(1)
>>> m.__getitem__(2)
>>> m.__getitem__(3)

>>> m[0]
>>> m[1]
>>> m[2]
>>> m[3]

>>> hasattr(m, '__iter__')
>>> hasattr(m, '__getitem__')
>>> it = iter(m)
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()

>>> list(m)

>>> for item in m:
```

```
...     print(item)
```

Optional Iterators

```
>>> m = [1, 2, 3]
>>> reversed(m)
>>> it = reversed(m)
>>> type(it)
>>> dir(it)
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()

>>> m
>>> for i in m:
...     print(i)

>>> m.__getitem__(0)
>>> m.__getitem__(1)
>>> m.__getitem__(2)
>>> m.__getitem__(3)

>>> it = reversed(m)
>>> it2 = it.__iter__()
>>> hasattr(it2, '__next__')

>>> m = [2 * i for i in range(3)]
>>> m
>>> type(m)

>>> mi = (2 * i for i in range(3))
>>> mi
>>> type(mi)
>>> hasattr(mi, '__next__')
>>> dir(mi)
>>> help(mi)
>>> mi.__next__()
>>> mi.__next__()
>>> mi.__next__()
>>> mi.__next__()
```

Optional Exercises: Iterators

```
>>> m = [1, 2, 3]
>>> it = iter(m)
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()

>>> for n in m:
...     print(n)

>>> d = {'one': 1, 'two': 2, 'three':3}
>>> it = iter(d)
>>> list(it)

>>> mi = (2 * i for i in range(3))
>>> list(mi)
>>> list(mi)

>>> import itertools
```

Take a look at the `itertools` module documentation.

```
>>> m = [1, 2, 3]
```

```
>>> it1 = iter(m)
>>> it2 = iter(it1)
>>> list(it1)
>>> list(it2)

>>> it1 = iter(m)
>>> it2 = iter(m)
>>> list(it1)
>>> list(it2)
>>> list(it1)
>>> list(it2)
```

Generators

```
>>> list_comprehension = [2 * i for i in range(5)]
>>> list_comprehension
>>> gen_exp = (2 * i for i in range(5))
>>> gen_exp
>>> hasattr(gen_exp, '__next__')
>>> list(gen_exp)
>>> list(gen_exp)

>>> for i in (2 * i for i in range(5)):
...     print(i)

>>> def list123():
...     yield 1
...     yield 2
...     yield 3

>>> list123
>>> list123()
>>> it = list123()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()

>>> for i in list123():
...     print(i)

>>> def even(limit):
...     for i in range(0, limit, 2):
...         print('Yielding', i)
...         yield i
...     print('done loop, falling out')

>>> it = even(3)
>>> it
>>> it.__next__()
>>> it.__next__()
>>> it.__next__()

>>> for i in even(3):
...     print(i)

>>> list(even(10))
```

Compare these versions

```
>>> def even_1(limit):
...     for i in range(0, limit, 2):
...         yield i

>>> def even_2(limit):
...     result = []
...     for i in range(0, limit, 2):
...         result.append(i)
...     return result

>>> [i for i in even_1(10)]
>>> [i for i in even_2(10)]
```

```
>>> def paragraphs(lines):
...     result = ''
...     for line in lines:
...         if line.strip() == '':
...             yield result
...             result = ''
...         else:
...             result += line
...     yield result

>>> list(paragraphs(open('eg.txt')))
>>> len(list(paragraphs(open('eg.txt'))))
```

First class objects

Python exposes many language features and places almost no constraints on what types data structures can hold.

Here's an example of using a dictionary of functions to create a simple calculator. In some languages this require a `case` or `switch` statement, or a series of `if` statements. If you've been using such a language for a while, this example may help you expand the range of solutions you can imagine in Python.

```
>>> 7+3
>>> import operator
>>> operator.add(7, 3)

>>> expr = '7+3'
>>> lhs, op, rhs = expr
>>> lhs, op, rhs
>>> lhs, rhs = int(lhs), int(rhs)
>>> lhs, op, rhs
>>> op, lhs, rhs
>>> operator.add(lhs, rhs)

>>> ops = {
...     '+': operator.add,
...     '-': operator.sub,
...     }

>>> ops[op] (lhs, rhs)

>>> def calc(expr):
...     lhs, op, rhs = expr
...     lhs, rhs = int(lhs), int(rhs)
...     return ops[op] (lhs, rhs)

>>> calc('7+3')
>>> calc('9-5')
>>> calc('8/2')
>>> ops['/'] = operator.truediv
>>> calc('8/2')

>>> class Unpacker:
...     slices = {
...         'first': slice(0, 3),
...         'hyde': slice(9, 12),
...         'myname': slice(18, 21)
...         }
...     #
...     def __init__(self, record):
...         self.record = record
...     #
...     def __getattr__(self, attr):
...         if attr in self.slices:
...             return self.record[self.slices[attr]]
...         raise AttributeError(
...             "'Unpacker' object has no attribute '{}'"
...             .format(attr))
...

>>> u = Unpacker('abcdefghijklmnopqrstuvwxyz')
```

```
>>> u.first
>>> u.hyde
>>> u.myname
```

Optional: Closures and partial functions

```
>>> def log(message, subsystem):
...     """
...     Write the contents of 'message'
...     to the specified subsystem.
...     """
...     print('LOG - {}: {}'.format(subsystem, message))

>>> log('Initializing server', 'server')
>>> log('Reading config file', 'server')

>>> def server_log(message):
...     log(message, 'server')

>>> server_log('Initializing server')
>>> server_log('Reading config file')

>>> import functools
>>> server_log = functools.partial(log, subsystem='server')

>>> server_log
>>> server_log.func is log
>>> server_log.keywords

>>> server_log('Initializing server')
>>> server_log('Reading config file')
```

Bound methods are a form of partials:

```
>>> SENTENCE_ENDING = '.?!'
>>> sentence = 'This is a sentence!'
>>> sentence[-1] in SENTENCE_ENDING
>>> '.' in SENTENCE_ENDING
>>> SENTENCE_ENDING.__contains__('.')
>>> SENTENCE_ENDING.__contains__(',')
>>> is_sentence_ending = SENTENCE_ENDING.__contains__
>>> is_sentence_ending('.')
>>> is_sentence_ending(',')
```

Yet another way to bind some data is to create a class and give it a `__call__` method:

```
>>> class SentenceEnding:
...     def __init__(self, characters):
...         self.punctuation = characters
...     #
...     def __call__(self, sentence):
...         return sentence[-1] in self.punctuation

>>> is_sentence1 = SentenceEnding('.')
>>> is_sentence1('This is a test.')
>>> is_sentence1('This is a test!')

>>> is_sentence2 = SentenceEnding('.!?')
>>> is_sentence2('This is a test.')
>>> is_sentence2('This is a test!')
```

Optional Exercises: namedtuple, operator

```
>>> import collections

>>> Month = collections.namedtuple(
...     'Month', 'name number days', verbose=True)

>>> jan = Month('January', 1, 31)
```

```
>>> jan.name
>>> jan[0]
>>> apr = Month('April', 3, 30)
>>> apr.days
>>> apr[2]
>>> jul = Month('July', 7, 31)

>>> m = [jan, apr, jul]

>>> def month_days(month):
...     return month.days

>>> import operator
>>> sorted(m, key=operator.itemgetter(0))
>>> sorted(m, key=operator.attrgetter('name'))
>>> sorted(m, key=operator.attrgetter('number'))
```

Evaluations