Python Epiphanies

Overview

This tutorial, presented at PyCon 2012 in San Jose by Stuart Williams (stuart@swilliams.ca), 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 almost exclusively use the interactive Python interpreter. I’ll be using Python 2.7 but most of this will work identically in 3.x.

Most exercise sections start out simple but increase quickly in difficulty in order to give more advanced students a challenge. We’ll move well before everyone has completed the entire section!

I am not providing the text of these exercises online because by typing them yourselves you will learn more.

Dictionaries and Namespaces

```
>>> month_number_to_name = [None, ’Jan’, ’Feb’, ’Mar’]          0
>>> month_number_to_name[1] # month 1 is January                1
>>> month_number_to_name[2] # month 2 is February               2
```

```
>>> month_name_to_number = {’Jan’: 1, ’Feb’: 2, ’Mar’: 3}       3
>>> month_name_to_number[’Jan’] # January is month 1            4
>>> month_name_to_number[’Feb’]  # February is month 2          5
```

```
>>> _namespace = {}                                             6
>>> _namespace                                                  7
>>> _namespace[’a’] = 7                                         8
>>> _namespace                                                  9
>>> _namespace[’a’] = 8                                         10
>>> _namespace                                                  11
>>> _namespace[’s’] = ’March’                                   12
>>> _namespace                                                  13
```

```
>>> a                                                           14
>>> dir()                                                       15
>>> a = 17
>>> a
>>> dir()
>>> s = 'March'
>>> dir()
>>> a
>>> del a
>>> dir()
>>> a
>>> del s
>>> dir()
```

Objects and Variables

Everything in Python is an object and has:

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

A single value:

```
>>> 1
>>> 1.0
>>> 'hello'
>>> (1, 2, 3)
>>> [1, 2, 3]
```

A single type:

```
>>> type(1)
>>> type(1.0)
>>> type('hello')
>>> type((1, 2, 3))
>>> type([1, 2, 3])
```

Some number of attributes:

```
>>> dir(1)
>>> (1).__doc__
>>> (1).__class__
>>> (1.0).__class__
>>> ('hello').__class__
>>> 'mississippi'.count
>>> 'mississippi'.count('s')
>>> (1, 2, 3).index
>>> [1, 2, 3].pop
```

A single id:

```
>>> id(1)
>>> id(1.0)
>>> id('hello')
>>> id((1, 2, 3))
>>> id([1, 2, 3])
```

Base classes:

```
>>> import inspect
>>> inspect.getmro(type('hello'))
```

```
>>> 'hello'.__class__
>>> type('hello') is 'hello'.__class__ is str
>>> 'hello'.__class__.__bases__
>>> 'hello'.__class__.__bases__[0]
>>> 'hello'.__class__.__bases__[0].__bases__
>>> inspect.getmro(type('hello'))
```

Exercises: Namespaces and Objects

Restart Python to unclutter the local namespace.

```
>>> dir()
>>> i = 1
>>> i
>>> dir()
>>> type(i)
>>> id(i)
>>> j = 1
>>> dir()
>>> id(j)
>>> id(i) == id(j)
>>> i is j
```

```
>>> m = [1, 2, 3]
>>> m
>>> n = m
>>> n
>>> dir()                                             74
>>> m is n                                            75
>>> m[1] = 'two'                                      76
>>> m                                                 77
>>> n                                                 78
```

```
>>> s = 'hello'                                       79
>>> s                                                 80
>>> id(s)                                             81
>>> s += ' there'                                     82
>>> s                                                 83
>>> id(s)                                             84
```

```
>>> m = [1, 2, 3]                                     85
>>> m                                                 86
>>> id(m)                                             87
>>> m += [4]                                          88
>>> m                                                 89
>>> id(m)                                             90
```

```
>>> int.__div__                                       91
>>> int.__div__ == int.__truediv__                    92
>>> int.__div__ = int.__truediv__                     93
```

```
>>> dir(None)                                         94
>>> dir(None) == dir(object)                          95
>>> dir(None) == dir(object())                        96
```

Namespaces

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

Assignment is a namespace operation, not 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. `s` instead of `_namespace['s']`.

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

The (*direct*) namespace search order is (from the python.org tutorial):

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

All namespace *changes* (assignment, `del`, `import`, `def`, `class`) happen in the local scope (i.e. in the current scope in which the namespace-changing code executes).

Let's look at some surprising behaviour:

```
>>> x = 1                                                                     97
>>> def test1():                                                              98
...     print('In test1 x == {}'.format(x))

>>> test1()                                                                   99


>>> def test2():                                                             100
...     x = 2
...     print('In test2 x == {}'.format(x))


>>> x                                                                        101
>>> test2()                                                                  102
>>> x                                                                        103


>>> def test3():                                                             104
...     print('In test3 x == {}'.format(x))
...     x = 3


>>> x                                                                        105
>>> test3()                                                                  106
>>> x                                                                        107


>>> test1.func_code.co_varnames                                              108
>>> test3.func_code.co_varnames                                              109


>>> def test4():                                                             110
...     global x
...     print('In test4 before, x == {}'.format(x))
...     x = 4
...     print('In test4 after, x == {}'.format(x))


>>> x                                                                        111
>>> test4()                                                                  112
>>> x                                                                        113

>>> test4.func_code.co_varnames                                              114
```

"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]

The Local Namespace

```
>>> help(dir)                                                     115
>>> dir()                                                         116
```

```
>>> import inspect                                                117
>>> from pprint import pprint                                     118
>>> pprint(inspect.getmembers(None))                              119
```

```
>>> # subtlety with exec, used by code.interactive                120
>>> __builtins__                                                  121
>>> type(__builtins__)                                            122
>>> __builtins__.keys()                                           123
```

```
>>> # To follow, you can do this:                                 124
>>> __my_builtins__ = __builtins__                                125
>>> # I fake it like this:                                        126
>>> __my_builtins__ = __import__(’__builtin__’)                   127
```

```
>>> from textwrap import fill                                     128
>>> def is_exception(s):                                          129
...     return ’Error’ in s or ’Warning’ in s
```

```
>>> print(fill(’ ’.join(                                          130
...     [b for b in dir(__my_builtins__)
...      if is_exception(b)]), 60))
... print(fill(’ ’.join(
...     [b for b in dir(__my_builtins__)
...      if not is_exception(b)]), 60))
```

Exercises: The Local Namespace

```
>>> locals().keys()                                               131
>>> globals().keys()                                              132
>>> locals() == globals()                                         133
>>> locals() is globals()                                         134
```

```
>>> x                                                             135
>>> locals()[’x’]                                                 136
>>> locals()[’x’] = 1                                             137
>>> locals()[’x’]                                                 138
>>> x                                                             139
>>> dir()                                                         140
```

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

- `bytes`
- `enumerate`, `reversed`
- `exit`, `help`, `license`, `quit`
- `True`, `False`, `None`, `NotImplemented`, `Ellipsis`

Namespace Changes

These change or modify a namespace:

- assignment
- `del`
- (`globals()` and `locals()`)
- `import`
- `def`
- `class`

Next we'll explore the last three.

```
>>> 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
```

```
>>> from pprint import pprint as pprint_function
>>> dir()
>>> pprint.pprint is pprint_function
>>> pprint
>>> pprint.pformat
```

```
>>> del pprint
>>> import pprint as pprint_module
>>> dir()
>>> pprint_module.pprint is pprint_function
>>> module_name = 'string'                                                     159
>>> string_module = __import__(module_name)                                    160
>>> string_module.uppercase                                                    161
>>> import string                                                              162
```

File structure:

```
folder1/
  file1.py

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

```
>>> dir()                                                                      163
>>> import folder1                                                             164
>>> import folder1.file1                                                       165
>>> import module1                                                             166
>>> dir()                                                                      167
>>> dir(module1)                                                               168
>>> import module1.file1                                                       169
>>> dir()                                                                      170
>>> dir(module1)                                                               171
>>> dir(module1.file1)                                                         172
>>> from module1 import file1                                                  173
>>> dir()                                                                      174
>>> dir(file1)                                                                 175
```

Exercise: The import statement

```
>>> import pprint                                                              176
>>> dir(pprint)                                                                177
>>> pprint.__doc__                                                             178
>>> pprint.__file__                                                            179
>>> pprint.__name__                                                            180
>>> pprint.__package__                                                         181
>>> dir(pprint)                                                                182
```

```
>>> from pprint import *                                                       183
>>> dir()                                                                      184
```

```
>>> reload(csv)                                                                185
>>> reload('csv')                                                              186
>>> import csv                                                                 187
>>> reload('csv')                                                              188
>>> reload(csv)                                                                189
>>> import sys                                                              190
>>> sys.path                                                                191
```

Functions

```
>>> def f(arg1, arg2, kwarg1=’kw1’, kwarg2=’kw2’,                           192
...        *args, **kwargs):
...     """
...     A function with regular and keyword arguments.
...     """
...     print(’arg1: {0}, arg2: {1}, ’
...        ’kwarg1: {2}, kwarg2: {3}’
...        .format(arg1, arg2, kwarg1, kwarg2))
...     if args:
...         print(’args:’, str(args))
...     if kwargs:
...         print(’kwargs:’, kwargs)
```

```
>>> f.__name__                                                              193
>>> dir()                                                                   194
>>> f.__name__ = ’g’                                                        195
>>> dir()                                                                   196
>>> f.__name__                                                              197
>>> f                                                                       198
>>> f.func_dict                                                             199
>>> f.foo = ’bar’                                                           200
>>> f.func_dict                                                             201

>>> f.func_defaults                                                         202
```

```
>>> f(1, 2)                                                                 203
>>> f(arg1=1, arg2=2)                                                       204
>>> f(arg2=1, arg1=2)                                                       205
>>> f(1, 2, 3)                                                              206
>>> f(1, 2, kwarg2=4)                                                       207
>>> f(1, kwarg1=3)                                                          208
>>> f(1, 2, 3, 4, 5, 6)                                                     209
>>> f(1, 2, 3, 4, keya=7, keyb=8)                                           210
>>> f(1, 2, 3, 4, 5, 6, keya=7, keyb=8)                                     211
```

Exercises: Functions

```
>>> def f(a1, a2, kw1=’k1’, kw2=’k2’):                                      212
...     print(repr((a1, a2, kw1, kw2)))
```

```
>>> f(1)
>>> f(1, 2)
>>> f(1, 2, 3)
>>> t = 1, 2
>>> t
>>> d = dict(kw1=3, kw2=4)
>>> d
>>> f(*t)
>>> f(**d)
>>> f(1, 2, **d)
>>> f(*t, **d)
```

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

Lists are mutable, strings are not

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

```
>>> old_s = s = ’hello’
>>> old_s, s, s is old_s, id(s), id(old_s)
>>> s = s + ’ there’
>>> old_s, s, s is old_s, id(s), id(old_s)
```

```
>>> old_s = s = ’hello’
>>> old_s, s, s is old_s, id(s), id(old_s)
>>> s += ’ there’
>>> old_s, s, s is old_s, id(s), id(old_s)
```

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

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

>>> # Why?
```

```
>>> import codeop, dis
>>> dis.dis(codeop.compile_command(’m = list(); m += 4’))
>>> dis.dis(codeop.compile_command("s = ’hello’; s += ’ there’"))
>>> m = [1, 2, 3]                                                                  248
>>> m                                                                              249
>>> m.__iadd__([4])                                                                250
>>> m                                                                              251
```

```
>>> # str.__iadd__ copies but list.__iadd__ mutates                                252
```

```
>>> # How are parameters passed?  Always by reference.                             253
```

```
>>> def test1(s):                                                                  254
...     print(’Before:’, s)
...     s += ’ there’
...     print(’After:’, s)
```

```
>>> str2 = ’hello’                                                                 255
>>> str2                                                                           256
>>> test1(str2)                                                                    257
>>> str2                                                                           258
>>> test1(’hello’)                                                                 259
```

```
>>> def test2(m):                                                                  260
...     print(’Before:’, m)
...     m += [4]
...     print(’After:’, m)
```

```
>>> list3 = [1, 2, 3]                                                              261
>>> list3                                                                          262
>>> test2(list3)                                                                   263
>>> list3                                                                          264
```

Decorators

```
>>> def square(n):                                                                 265
...     return n * n
```

```
>>> square(2)                                                                      266
>>> square(3)                                                                      267
```

```
>>> def stringify(func):                                                           268
...     def convert_to_str(n):
...         return str(func(n))
...     return convert_to_str
```

```
>>> def stringify(func):                                        269
...     def convert_to_str(n):
...         print(’called convert_to_str({})’
...               .format(n))
...         return str(func(n))
...     print(’called stringify({})’
...           .format(func))
...     return convert_to_str
```

```
>>> square                                                      270
>>> square_str = stringify(square)                              271
>>> square_str                                                  272
>>> square_str(3)                                               273
```

```
>>> @stringify                                                  274
>>> def cube(n):                                                275
...     return n * n * n

>>> cube(2)                                                     276
```

Exercises: changing the local namespace

A decorator is a function that takes function as a parameter and *usually* returns a function, but doesn’t have to. What does the following code do?

Restart Python to unclutter the local namespace.

```
>>> dir()                                                       277
>>> x                                                           278
```

```
>>> def value(f):                                               279
...     return f()
```

```
>>> @value                                                      280
>>> def x():                                                    281
...     return 1
```

```
>>> dir()                                                       282
>>> x                                                           283
>>> type(x)                                                     284
```

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

```
>>> del x                                                                  285
>>> x                                                                      286
>>> def x():                                                               287
...     return 1
```

```
>>> x                                                                      288
>>> x = value(x)                                                           289
>>> x                                                                      290
```

The class statement

Remember, everything in Python is an object and has:

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

Many objects are instances of classes. The type of an object is its class.

Classes are instances of *metaclasses*. The type of a class is a metaclass, i.e. `type(type(anObject))` is a metaclass.

Are classes and metaclasses objects?

1. The `class` statement creates a new namespace and all its name assignments (e.g. function definitions) are bound to the class object.

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

`ClassName.__init__(<new object>, ...)` is called automatically, passing in the new object which was already created (by `__new__`).

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

```
>>> class Num(object):                                                     291
...     def __init__(self, amount):
...         self.amount = amount
...     #
...     def add(self, value):
...         return self.amount + value
```

```
>>> Num                                                                    292
>>> Num.__init__                                                           293
>>> Num.add                                                                294
>>> dir(Num)                                                               295
>>> num2 = Num(2)                                                        296
>>> num2.amount                                                          297
>>> num2                                                                 298
>>> num2.__init__                                                        299
>>> num2.add                                                             300
>>> dir(num2)                                                            301
>>> num2.__dict__                                                        302
>>> Num.__dict__                                                         303
```

```
>>> num2.add                                                             304
>>> num2.add(3)                                                          305
>>> Num.add                                                              306
>>> Num.add(2)                                                           307
>>> Num.add(2, 3)                                                        308
>>> Num.add(num2, 3)                                                     309

>>> num2.add(3)                                                          310
```

```
>>> def set_amount_double(self, amount):                                 311
...     self.amount = 2 * amount
```

```
>>> Num.__init__                                                         312
>>> help(Num.__init__)                                                   313
>>> Num.__init__ = set_amount_double                                     314
>>> Num.__init__                                                         315
>>> help(Num.__init__)                                                   316
```

```
>>> num4 = Num(2)                                                        317
>>> num4.add(5)                                                          318
>>> num2.__init__                                                        319
```

```
>>> def multiply_by(num, value):                                         320
...     return num.amount * value
```

```
>>> # Methods live in classes, not instances.                            321
>>> # Let's make a mistake.                                              322
>>> num4.mul = multiply_by                                               323
>>> num4.mul                                                             324
>>> num4.mul(5)                                                          325
>>> num4.mul(num4, 5)                                                    326
```

```
>>> num5 = Num(5)                                                        327
>>> num5.mul                                                             328
>>> dir(num4)                                                                329
>>> dir(Num)                                                                 330
>>> Num.mul = multiply_by                                                    331
>>> num4.mul(5)                                                              332
>>> num5.mul(5)                                                              333
>>> dir(num4)                                                                334
>>> num4.mul                                                                 335
>>> del num4.mul                                                             336
>>> Num.mul                                                                  337
>>> num4.mul                                                                 338
>>> num4.mul(5)                                                              339
```

```
>>> num4                                                                     340
>>> num4.mul                                                                 341
>>> dir(num4.mul)                                                            342
>>> num4.mul.im_class                                                        343
```

```
>>> num4.mul.__self__                                                        344
>>> num4.mul.__self__ is num4                                                345
>>> num4.mul.__self__ is num4.mul.im_self                                    346
>>> num4.mul.__func__                                                        347
>>> num4.mul.__func__ is multiply_by                                         348
>>> num4.mul.__func__ is num4.mul.im_func                                    349
>>> help(num4.mul.__func__)                                                  350
```

```
>>> num4.mul(5)                                                              351
>>> num4.mul.__func__(num4.mul.__self__, 5)                                  352
```

Exercises: The class statement

Type in this class statement:

```
>>> class Prefixer(object):                                                  353
...     pass
```

Now at the interactive prompt, similar to the demonstration above, incrementally add the method(s) required to make the following code work:

```
>>> arrow = Prefixer('-> ')                                                  354
>>> assert arrow.prepend(['line 1', 'line2']) == ['-> line 1', '-> line 2']  355
```

The type function for classes

Glyph Lefkowitz in “Turtles All The Way Down...” at PyCon 2010:

The class statement is just a way to call a function, take the result, and put it into a namespace.

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

```
>>> print(type.__doc__)                                                                    356
>>> DoubleNum = type(                                                                      357
...     'DoubleNum',
...     (object,),
...     { '__init__': set_amount_double,
...       'mul': multiply_by,
...     })
```

```
>>> num6 = DoubleNum(3)                                                                    358
>>> type(num6)                                                                             359
>>> num6.__class__                                                                         360
>>> num6.__dict__                                                                          361
>>> num6.amount                                                                            362
>>> num6.mul(4)                                                                            363
```

This dynamic call to type is what the `class` statement actually triggers.

However, “When the class definition is read, if __*metaclass*__ is defined then the callable assigned to it will be called instead of `type()`.”

`__metaclass__` “can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`.” [Language Reference section 3.4.3]

Exercise: The class statement

What does the following do? Use only one of the “2.7” and “3.x” definitions of `class x`.

```
>>> def one(name, bases, attrs):                                                           364
...     return 1
```

```
>>> class x:  # Python 2.7 syntax                                                          365
...     __metaclass__ = one # call this to create the class
```

OR

```
>>> class x(metaclass=one):  # Python 3.x syntax                                           366
...     pass
```

```
>>> x                                                                                      367
```

What does this code do?

```
>>> def two(klass):                                                        368
...     return 2
```

```
>>> @two                                                                   369
>>> class y(object):                                                       370
...     pass
```

```
>>> y                                                                      371
```

Standard class methods

- `__new__`, `__init__`, `__del__`, `__repr__`, `__str__`, `__format__`
- `__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__call__`, `__dir__`
- `__len__`, `__getitem__`, `__missing__`, `__setitem__`, `__delitem__`, `__contains__`, `__iter__`
- `__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__`, `__next__`

```
>>> class UpperAttr(object):                                               372
...     """
...     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()                                                        373
>>> d.__dict__                                                             374
>>> d.foo = 'bar'                                                          375
>>> d.foo                                                                  376
>>> d.__dict__                                                             377
>>> d.FOO                                                                  378
>>> d.bar                                                                  379
```

Exercise: Standard class methods

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

```
>>> class Get(object):                                                    380
...     def __getitem__(self, key):
...         print(’called __getitem__({} {})’
...             .format(type(key), repr(key)))
```

```
>>> g = Get()                                                             381
>>> g[1]                                                                  382
>>> g[-1]                                                                 383
>>> g[0:3]                                                                384
>>> g[0:10:2]                                                             385
>>> g[’Jan’]                                                              386
>>> g[g]                                                                  387
```

```
>>> m = list(’abcdefghij’)                                                388
>>> m[0]                                                                  389
>>> m[-1]                                                                 390
>>> m[::2]                                                                391
>>> s = slice(3)                                                          392
>>> m[s]                                                                  393
>>> m[slice(1, 3)]                                                        394
>>> m[slice(0, 2)]                                                        395
>>> m[slice(0, len(m), 2)]                                                396
>>> m[::2]                                                                397
```

Properties

```
>>> class PropertyExample(object):                                        398
...     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()                                                 399
```

```
>>> p.setx(’foo’)                                                         400
>>> p.getx()                                                    401
>>> p.x = ’bar’                                                 402
>>> p.x                                                         403
>>> del p.x                                                     404
```

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__()`, calls it starting at zero, and handles `IndexError` by raising `StopIteration`.

```
>>> class MyList(object):                                       405
...     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’])                                 406


>>> m.__getitem__(0)                                            407
>>> m.__getitem__(1)                                            408
>>> m.__getitem__(2)                                            409
>>> m.__getitem__(3)                                            410


>>> m[0]                                                        411
>>> m[1]                                                        412
>>> m[2]                                                        413
>>> m[3]                                                        414


>>> hasattr(m, ’__iter__’)                                      415
>>> hasattr(m, ’__getitem__’)                                   416
>>> it = iter(m)                                                417
>>> it.next()                                                   418
>>> it.next()                                                   419
>>> it.next()                                                   420
>>> it.next()                                                   421

>>> list(m)                                                     422
>>> for item in m:                                                     423
...     print(item)


>>> m = MyList({’Jan’: 1, ’Feb’: 2, ’Mar’: 3})                         424
>>> m[’Jan’]                                                           425
>>> for item in m:                                                     426
...     print(m)

>>> list(m)                                                            427


>>> m = [1, 2, 3]                                                      428
>>> reversed(m)                                                        429
>>> it = reversed(m)                                                   430
>>> type(it)                                                           431
>>> dir(it)                                                            432
>>> it.next()                                                          433
>>> it.next()                                                          434
>>> it.next()                                                          435
>>> it.next()                                                          436
>>> it.next()                                                          437
>>> it.next()                                                          438


>>> m                                                                  439
>>> for i in m:                                                        440
...     print(i)


>>> m.next()                                                           441
>>> it = iter(m)                                                       442
>>> it.next()                                                          443
>>> it.next()                                                          444
>>> it.next()                                                          445
>>> it.next()                                                          446


>>> m.__getitem__(0)                                                   447
>>> m.__getitem__(1)                                                   448
>>> m.__getitem__(2)                                                   449
>>> m.__getitem__(3)                                                   450


>>> it = reversed(m)                                                   451
>>> it2 = it.__iter__()                                                452
>>> hasattr(it2, ’next’)                                               453


>>> m = [2 * i for i in range(3)]                                      454
>>> m                                                                  455
>>> type(m)                                                            456
>>> mi = (2 * i for i in range(3))                                    457
>>> mi                                                                458
>>> type(mi)                                                          459
>>> hasattr(mi, 'next')                                               460
>>> dir(mi)                                                           461
>>> help(mi)                                                          462
>>> mi.next()                                                         463
>>> mi.next()                                                         464
>>> mi.next()                                                         465
>>> mi.next()                                                         466
```

Exercises: Iterators

```
>>> m = [1, 2, 3]                                                     467
>>> it = iter(m)                                                      468
>>> it.next()                                                         469
>>> it.next()                                                         470
>>> it.next()                                                         471
>>> it.next()                                                         472
```

```
>>> for n in m:                                                       473
...     print(n)
```

```
>>> it = iter(m)                                                      474
>>> it2 = iter(it)                                                    475
>>> list(it2)                                                         476
>>> list(it)                                                          477
```

```
>>> it1 = iter(m)                                                     478
>>> it2 = iter(m)                                                     479
>>> list(it1)                                                         480
>>> list(it2)                                                         481
>>> list(it1)                                                         482
>>> list(it2)                                                         483
```

```
>>> d = {'one': 1, 'two': 2, 'three':3}                               484
>>> it = iter(d)                                                      485
>>> list(it)                                                          486
```

```
>>> mi = (2 * i for i in range(3))                                    487
>>> list(mi)                                                          488
>>> list(mi)                                                          489

>>> import itertools                                                  490
```

Take a look at the `itertools` module documentation.

Iterators continued

```
>>> class MyIterable(object):                               491
...     pass

>>> myit = MyIterable()                                     492
>>> iter(myit)                                              493

>>> def mygetitem(self, key):                               494
...     # Note we ignore self!
...     print('called mygetitem({})'.format(key))
...     return [0, 1, 2][key]

>>> MyIterable.__getitem__ = mygetitem                      495
>>> iter(myit)                                              496
>>> list(iter(myit))                                        497

>>> 1 in myit                                               498
>>> x, y, z = myit                                          499

>>> myit2 = iter([1, 2, 2, 3])                              500
>>> 2 in myit2                                              501
>>> 2 in myit2                                              502
>>> 2 in myit2                                              503

>>> class ListOfThree(object):                              504
...     def __iter__(self):
...         self.index = 0
...         return self
...     #
...     def next(self):
...         if self.index < 3:
...             self.index += 1
...             return self.index
...         raise StopIteration

>>> m3 = ListOfThree()                                      505
>>> m3it = iter(m3)                                         506
>>> m3it.next()                                             507
>>> m3it.next()                                             508
>>> m3it.next()                                             509
>>> m3it.next()                                             510

>>> list(m3it)                                              511
>>> list(m3it)                                              512
```

Exercises: Iterators continued

Design a subclass of dict whose iterator would return its keys, as does `dict`, but in sorted order, and without using `yield`.

Design a class `reversed` to mimic Python's built in `reverse` function. Assume an indexable sequence as parameter.

Implement one or both of these designs.

Generators

```
>>> gen_exp = (2 * i for i in range(5))                                    513
>>> gen_exp                                                                514
>>> hasattr(gen_exp, 'next')                                               515
>>> list(gen_exp)                                                          516
>>> list(gen_exp)                                                          517
```

```
>>> for i in (2 * i for i in range(5)):                                    518
...     print(i)
```

```
>>> def list123():                                                         519
...     yield 1
...     yield 2
...     yield 3
```

```
>>> list123                                                                520
>>> list123()                                                              521
>>> it = list123()                                                         522
>>> it.next()                                                              523
>>> it.next()                                                              524
>>> it.next()                                                              525
>>> it.next()                                                              526
```

```
>>> def even(limit):                                                       527
...     for i in range(0, limit, 2):
...         print('Yielding', i)
...         yield i
...     print('done loop, falling out')
```

```
>>> it = even(3)                                                           528
>>> it                                                                     529
>>> it.next()                                                              530
>>> it.next()                                                              531
>>> it.next()                                                              532
>>> for i in even(3):                                                533
...     print(i)

>>> list(even(10))                                                   534
```

Compare these versions

```
>>> def even_1(limit):                                               535
...     for i in range(0, limit, 2):
...         yield i
```

```
>>> def even_2(limit):                                               536
...     result = []
...     for i in range(0, limit, 2):
...         result.append(i)
...     return result
```

```
>>> [i for i in even_1(10)]                                          537
>>> [i for i in even_2(10)]                                          538
```

```
>>> def paragraphs(lines):                                           539
...     result = ’’
...     for line in lines:
...         if line.strip() == ’’:
...             yield result
...             result = ’’
...         else:
...             result += line
...     yield result
```

```
>>> list(paragraphs(open(’eg.txt’)))                                 540
>>> len(list(paragraphs(open(’eg.txt’))))                            541
```

Exercises: Generators

Write a generator `sdouble(str)` that takes a string a returns that string “doubled” 5 times. E.g. `sdbouble(’s’)` would yield these values: `[’s’, ’ss’, ’ssss’, ’ssssssss’, ’ssssssssssssssss’]`.

Re-design the earlier (iterator subclass of `dict`) exercise to use `yield` in the `next` method.

Write a generator that returns sentences out of a paragraph. Make some simple assumptions about how sentences start and/or end.

Write code which reads a file and produces a histogram of the frequency of all words in the file.

Explore further the `itertools` module.

First class objects

Python exposes almost all of the language for you to hack.

- slices
- functions
- classes
- etc.

This is very powerful for library authors, like you.

Here’s an example of functions as first class objects to create a simple calculator.

```
>>> 7+3                                                 542
>>> import operator                                     543
>>> operator.add(7, 3)                                  544
```

```
>>> expr = ’7+3’                                        545
>>> lhs, op, rhs = expr                                 546
>>> lhs, op, rhs                                        547
>>> lhs, rhs = int(lhs), int(rhs)                       548
>>> lhs, op, rhs                                        549
>>> op, lhs, rhs                                        550
>>> operator.add(lhs, rhs)                              551
```

```
>>> ops = {                                             552
...     ’+’: operator.add,
...     ’-’: operator.sub,
...     }

>>> ops[op] (lhs, rhs)                                  553
```

```
>>> def calc(expr):                                     554
...     lhs, op, rhs = expr
...     lhs, rhs = int(lhs), int(rhs)
...     return ops[op] (lhs, rhs)
```

```
>>> calc(’7+3’)                                         555
>>> calc(’9-5’)                                         556
>>> calc(’8/2’)                                         557
>>> ops[’/’] = operator.div                             558
>>> calc(’8/2’)                                         559
>>> class Unpacker(object):                                                          560
...     slices = {
...         'header': slice(0, 3),
...         'trailer': slice(12, 18),
...         'middle': slice(6, 9)
...         }
...     #
...     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.header                                                                         561
>>> u.trailer                                                                        562
>>> u.middle                                                                         563
```

Partial functions and closures

```
>>> def log(message, subsystem):                                                     564
...     """
...     Write the contents of 'message'
...     to the specified subsystem.
...     """
...     print('LOG - {}: {}'.format(subsystem, message))
```

```
>>> log('Initializing server', 'server')                                             565
>>> log('Reading config file', 'server')                                             566
```

```
>>> import functools                                                                 567
>>> server_log = functools.partial(log, subsystem='server')                          568
```

```
>>> server_log                                                                       569
>>> server_log.func is log                                                           570
>>> server_log.keywords                                                              571
```

```
>>> server_log('Initializing server')                                                572
>>> log('Initializing server', 'server')                                             573
>>> server_log('Reading config file')                                                574
>>> log('Reading config file', 'server')                                             575
>>> def client_log(message):                                                   576
...     log(message, 'client')
```

```
>>> client_log('Initializing client')                                          577
>>> log('Initializing client', 'client')                                       578
```

Exercise: namedtuple, operator

```
>>> import collections                                                         579
```

```
>>> Month = collections.namedtuple('Month', 'name number days', verbose=True)  580
```

```
>>> jan = Month('January', 1, 31)                                              581
>>> jan.name                                                                   582
>>> jan[0]                                                                     583
>>> apr = Month('April', 3, 30)                                                584
>>> jul = Month('July', 7, 31)                                                 585
```

```
>>> m = [jan, apr, jul]                                                        586
```

```
>>> import operator                                                            587
>>> sorted(m, key=operator.itemgetter(0))                                      588
>>> sorted(m, key=operator.attrgetter('name'))                                 589
>>> sorted(m, key=operator.attrgetter('number'))                               590
```