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.

License: This PyCon 2012 Python Epiphanies Tutorial by Stuart Williams is licensed under a Creative Commons Attribution-Share Alike 2.5 Canada License (http://creativecommons.org/licenses/by-sa/2.5/ca/).

Dictionaries and Namespaces

```python
>>> month_number_to_name = [None, 'Jan', 'Feb', 'Mar']
>>> 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}
>>> month_name_to_number['Jan'] # January is month 1
4
>>> month_name_to_number['Feb'] # February is month 2
5
```

```python
>>> _namespace = {}
>>> _namespace['a'] = 7
>>> _namespace['a'] = 8
>>> _namespace['s'] = 'March'
```

```python
>>> a
```

```python
>>> 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:

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

A single type:

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

Some number of attributes:

```python
>>> dir(1)
>>> (1).__doc__
>>> (1).__class__
>>> (1.0).__class__
```
A single id:

```python
>>> id(1) 46
>>> id(1.0) 47
>>> id('hello') 48
>>> id((1, 2, 3)) 49
>>> id([1, 2, 3]) 50
```

Base classes:

```python
>>> import inspect 51
>>> inspect.getmro(type('hello')) 52
```

```python
>>> 'hello'.__class__ 53
>>> type('hello') is 'hello'.__class__ is str 54
>>> 'hello'.__class__.__bases__ 55
>>> 'hello'.__class__.__bases__[0] 56
>>> 'hello'.__class__.__bases__[0].__bases__ 57
>>> inspect.getmro(type('hello')) 58
```

Exercises: Namespaces and Objects

Restart Python to unclutter the local namespace.

```python
>>> dir() 59
>>> i = 1 60
>>> i 61
>>> dir() 62
>>> type(i) 63
>>> id(i) 64
>>> j = 1 65
>>> dir() 66
>>> id(j) 67
>>> id(i) == id(j) 68
>>> i is j 69
```

```python
>>> m = [1, 2, 3] 70
>>> m 71
>>> n = m 72
>>> n 73
```
Namespaces

A namespace is a mapping from 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. 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:

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

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

>>> x
>>> test2()
>>> x

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

>>> x
>>> test3()
>>> x

>>> test1.func_code.co_varnames
>>> test3.func_code.co_varnames

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

>>> x
>>> test4()
>>> x

>>> test4.func_code.co_varnames
```

“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

```python
>>> help(dir)
```

```python
>>> dir()
```

```python
>>> import inspect
>>> from pprint import pprint
>>> pprint(inspect.getmembers(None))
```

```python
>>> # subtlety with exec, used by code.interactive
>>> __builtins__
>>> type(__builtins__)
>>> __builtins__.keys()
```

```python
>>> # To follow, you can do this:
>>> __my_builtins__ = __builtins__
>>> # I fake it like this:
>>> __my_builtins__ = __import__('__builtin__')
```

```python
>>> from textwrap import fill
def is_exception(s):
...   return 'Error' in s or 'Warning' in s
```

```python
>>> print(fill(' '.join([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

```python
>>> locals().keys()
>>> globals().keys()
>>> locals() == globals()
>>> locals() is globals()
```

```python
>>> 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 via introspection (e.g. `type`, `inspect.getmro`, and `help`) or the Python documentation some of the following surprising 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.

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

```python
>>> pprint.pformat
>>> pprint.pprint
>>> pprint.foo
```

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

```python
>>> del pprint
>>> import pprint as pprint_module
>>> dir()
>>> pprint_module.pprint is pprint_function
```

>>> module_name = 'string'
>>> string_module = __import__(module_name)
>>> string_module.uppercase

File structure:

tool1/
  file1.py

tool1/__init__.py -- zero length
file1.py:
  attribute1 = 1

>>> dir()
>>> import tool1
>>> import tool1.file1
>>> import tool1.__init__
>>> dir()
>>> dir(tool1)
>>> dir(tool1.file1)
>>> import tool1.__init__
>>> dir()
>>> dir(tool1)
>>> dir(tool1.file1)
>>> from tool1 import file1
>>> dir()
>>> dir(file1)

Exercise: The import statement

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

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

>>> reload(csv)
>>> reload('csv')
>>> import csv
>>> reload('csv')
>>> reload(csv)
Functions

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

```python
>>> f.__name__
>>> dir()
>>> f.__name__ = 'g'
>>> dir()
>>> f.__name__
>>> f
>>> f.func_dict
>>> f.foo = 'bar'
>>> f.func_dict
>>> f.func_defaults
>>> f(1, 2)
>>> f(arg1=1, arg2=2)
>>> f(arg2=1, arg1=2)
>>> f(1, 2, 3)
>>> f(1, 2, kwarg2=4)
>>> f(1, kwarg1=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)
```

Exercises: Functions

```python
>>> def f(a1, a2, kw1='k1', kw2='k2'):
    ...     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]
>>> m
>>> m.__iadd__([4])
>>> m

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

>>> # How are parameters passed? Always 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

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

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

>>> def stringify(func):
...     def convert_to_str(n):
...         return str(func(n))
...     return convert_to_str
>>> def stringify(func):
...     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
>>> square_str = stringify(square)
>>> square_str
>>> square_str(3)

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

>>> cube(2)

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()
>>> x

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

>>> @value
>>> def x():
...     return 1

>>> dir()
>>> x
>>> type(x)

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

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

```python
>>> Num
>>> Num.__init__
>>> Num.add
>>> dir(Num)
```
>>> num2 = Num(2)
>>> num2.amount
>>> num2
>>> num2.__init__
>>> num2.add
>>> dir(num2)
>>> num2.__dict__
>>> Num.__dict__

>>> num2.add
>>> num2.add(3)
>>> Num.add
>>> Num.add(2)
>>> Num.add(2, 3)
>>> Num.add(num2, 3)

>>> num2.add(3)

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

>>> Num.__init__
>>> help(Num.__init__)  
>>> Num.__init__ = set_amount_double
>>> Num.__init__
>>> help(Num.__init__)  

>>> num4 = Num(2)
>>> num4.add(5)
>>> num2.__init__

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

>>> # Methods live in classes, not instances.
>>> # Let's make a mistake.
>>> num4.mul = multiply_by
>>> num4.mul
>>> num4.mul(5)
>>> num4.mul(num4, 5)

>>> num5 = Num(5)
>>> num5.mul
>>> dir(num4)
>>> dir(Num)
>>> Num.mul = multiply_by
>>> num4.mul(5)
>>> num5.mul(5)
>>> dir(num4)
>>> num4.mul
>>> del num4.mul
>>> Num.mul
>>> num4.mul
>>> num4.mul(5)

>>> num4
>>> num4.mul
>>> dir(num4.mul)
>>> num4.mul.im_class

>>> num4.mul.__self__
>>> num4.mul.__self__ is num4
>>> num4.mul.__self__ is num4.mul.im_self
>>> num4.mul.__func__
>>> num4.mul.__func__ is multiply_by
>>> num4.mul.__func__ is num4.mul.im_func
>>> help(num4.mul.__func__)

>>> num4.mul(5)
>>> num4.mul.__func__(num4.mul.__self__, 5)

Exercises: The class statement

Type in this class statement:

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

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

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

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.

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

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

>>> num6 = DoubleNum(3)
>>> type(num6)
>>> num6.__class__
>>> num6.__dict__
>>> num6.amount
>>> num6.mul(4)
```

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.

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

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

OR

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

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

>>> @two
>>> class y(object):
...     pass

>>> y

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__,
  __ior__, __ilshift__, __irshift__
- __int__, __long__, __float__, __complex__, __oct__, __hex__, __coerce__
- __radd__, __rsub__, __rmul__, __rdiv__, etc.
- __enter__, __exit__, __next__

>>> class UpperAttr(object):
...     ""
...     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()
>>> d.__dict__
>>> d.foo = 'bar'
>>> d.foo
>>> d.__dict__
>>> d.FOO
>>> d.bar

17
Exercise: Standard class methods

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

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

```python
>>> g = Get()
>>> g[1]
>>> g[-1]
>>> g[0:3]
>>> g[0:10:2]
>>> g['Jan']
>>> g[g]
```

```python
>>> m = list('abcdefghij')
>>> m[0]
>>> m[-1]
>>> m[:-1]
>>> m[:2]
>>> s = slice(3)
>>> m[s]
>>> m[slice(1, 3)]
>>> m[slice(0, 2)]
>>> m[slice(0, len(m), 2)]
>>> m[:2]
```

Properties

```python
>>> class PropertyExample(object):
...     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.")
```

```python
>>> p = PropertyExample()
>>> p.setx('foo')
```

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.

```python
>>> class MyList(object):
...     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'])
```

```python
>>> m.__getitem__(0)
>>> m.__getitem__(1)
>>> m.__getitem__(2)
>>> m.__getitem__(3)
>>> m[0]
>>> m[1]
>>> m[2]
>>> m[3]
```

```python
>>> hasattr(m, '__iter__')
>>> hasattr(m, '__getitem__')
>>> it = iter(m)
>>> it.next()
>>> it.next()
>>> it.next()
>>> it.next()
```

```python
>>> list(m)
```
>>> for item in m:
...     print(item)

>>> m = MyList({'Jan': 1, 'Feb': 2, 'Mar': 3})
>>> m['Jan']

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

>>> list(m)

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

>>> m

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

>>> m.next()
>>> it = iter(m)
>>> it.next()
>>> it.next()
>>> it.next()

>>> 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)
Exercises: Iterators

```python
tests = [1, 2, 3]
for t in tests:
    print(t)
tests2 = iter(tests)
list(tests2)
```
Iterators continued

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

>>> myit = MyIterable()
>>> iter(myit)

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

>>> MyIterable.__getitem__ = mygetitem
>>> iter(myit)
>>> list(iter(myit))

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

>>> myit2 = iter([1, 2, 2, 3])
>>> 2 in myit2
>>> 2 in myit2
>>> 2 in myit2

>>> class ListOfThree(object):
...     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()
>>> m3it = iter(m3)
>>> m3it.next()
>>> m3it.next()
>>> m3it.next()
>>> m3it.next()

>>> list(m3it)
>>> list(m3it)
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

```python
>>> gen_exp = (2 * i for i in range(5))
>>> gen_exp
... yield 1
... yield 2
... yield 3
```

```python
>>> list123():
... yield 1
... yield 2
... yield 3
```

```python
>>> list123
>>> list123()
>>> it = list123()
>>> it.next()
>>> it.next()
>>> it.next()
```

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

```python
>>> it = even(3)
...     for i in range(0, limit, 2):
...         print('Yielding', i)
...         yield i
...     print('done loop, falling out')
```
>>> 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'))))

Exercises: Generators

Write a generator sdouble(str) that takes a string a returns that string “doubled” 5 times. E.g. sddouble('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.

```python
>>> 7 + 3
5
>>> import operator
>>> operator.add(7, 3)
5

>>> expr = '7+3'
>>> lhs, op, rhs = expr
>>> lhs, op, rhs
5

>>> lhs, rhs = int(lhs), int(rhs)
>>> op, lhs, rhs
5

>>> operator.add(lhs, rhs)
5

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

>>> ops[op](lhs, rhs)
5

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

>>> calc('7+3')
5
>>> calc('9-5')
5
>>> calc('8/2')
5

>>> ops['/'] = operator.div
>>> calc('8/2')
5
```
>>> class Unpacker(object):
...     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
>>> u.trailer
>>> u.middle

Partial functions and closures

>>> 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')

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

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

>>> server_log('Initializing server')
>>> log('Initializing server', 'server')
>>> server_log('Reading config file')
>>> log('Reading config file', 'server')
>>> def client_log(message):
...     log(message, 'client')

>>> client_log('Initializing client')
>>> log('Initializing client', 'client')

Exercise: 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)
>>> jul = Month('July', 7, 31)

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

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