Python Epiphanies

Introduction

Python Epiphanies Tutorial at PyData NYC 2015

- http://bit.ly/pydatanyc-epiphanies (http://bit.ly/pydatanyc-epiphanies)
- Stuart Williams
- stuart@swilliams.ca
- @ceilous

- Tutorial for intermediate Python users
- Mostly one-liners
- You can use REPL, don't need IPython or Jupyter
- Python 3.4 preferred, 2.x OK
- Errors are intentional
- I recommend the HTML file

1 Objects

1.1 Back to the Basics: Objects

Let's go back to square one and be sure we understand the basics about objects in Python.

Objects can be created via literals.

In [ ]:

```
1
```

In [ ]:

```
3.14
```

In [ ]:

```
3.14j
```

In [ ]:

```
'a string literal'
```

In [ ]:

```
b'a bytes literal'
```

In [ ]:

```
(1, 2)
```

In [ ]:

```
[1, 2]
```

In [ ]:

```
{'one': 1, 'two': 2}
```

In [ ]:

```
{'one', 'two'}
```

Some constants are created on startup and have names.

In [ ]:

```
False, True
```

In [ ]:

```
None
```

In [ ]:

```
NotImplemented, Ellipsis
```

There are also some built-in types and functions.

In [ ]:

```
int, list
```

In [ ]:

```
any, len
```

Everything (*everything*) in Python (at runtime) is an object.

Every object has:

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

Let's explore each of these in turn.

Every object has a single type.

In [ ]:

```
type(1)
```

In [ ]:

```
type(3.14)
```

In [ ]:

```
type(3.14j)
```

In [ ]:

```
type('a string literal')
```

In [ ]:

```
type(b'a bytes literal')
```

In [ ]:

```
type((1, 2))
```

In [ ]:

```
type([1, 2])
```

In [ ]:

```
type({'one': 1, 'two': 2})
```

In [ ]:

```
type({'one', 'two'})
```

In [ ]:

```
type(True)
```

In [ ]:

```
type(None)
```

Every object has some number of attributes.

In [ ]:

```
True.__doc__
```

In [ ]:

```
'a string literal'.__add__
```

In [ ]:

```
callable('a string literal'.__add__)
```

In [ ]:

```
'a string literal'.__add__('!')
```

Every object has one or more base classes, accessible via attributes.

In [ ]:

```
True.__class__
```

In [ ]:

```
True.__class__.__bases__
```

In [ ]:

```
True.__class__.__bases__[0]
```

In [ ]:

```
True.__class__.__bases__[0].__bases__[0]
```

The method resolution order for classes is stored in `__mro__` by the class's `mro` method, which can be overridden.

In [ ]:

```
bool.__mro__
```

In [ ]:

```
import inspect
```

In [ ]:

```
inspect.getmro(True)
```

In [ ]:

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

In [ ]:

```
inspect.getmro(type(3))
```

In [ ]:

```
inspect.getmro(type('a string literal'))
```

Every object has a single unique id, which in CPython is a memory address.

In [ ]:

```
id(3)
```

In [ ]:

```
id(3.14)
```

In [ ]:

```
id('a string literal')
```

In [ ]:

```
id(True)
```

We can create objects by calling other callable objects (usually functions, methods, and classes).

In [ ]:

```
len
```

In [ ]:

```
callable(len)
```

In [ ]:

```
len('a string literal')
```

In [ ]:

```
'a string literal'.__len__
```

In [ ]:

```
'a string literal'.__len__()
```

In [ ]:

```
callable(int)
```

In [ ]:

```
int(3.14)
```

In [ ]:

```
int()
```

In [ ]:

```
dict
```

In [ ]:

```
dict()
```

In [ ]:

```
dict(pi=3.14, e=2.71)
```

In [ ]:

```
callable(True)
```

In [ ]:

```
True()
```

In [ ]:

```
bool()
```

1.2 Instructions for Completing Exercises

- Most sections include a set of exercises.
- Sometimes they reinforce learning
- Sometimes they introduce new material.
- Within each section exercises start out easy and get progressively harder.
- To maximize your learning:
  - Type the code in yourself instead of copying and pasting it.
  - Before you hit Enter try to predict what Python will do.
- A few of the exercises have intentional typos or code that is supposed to raise an exception. See what you can learn from them.
- Don't worry if you get stuck - I will go through the exercises and explain them in the video.

1.3 Exercises: Objects

In [ ]:

```
5.0
```

In [ ]:

```
dir(5.0)
```

In [ ]:

```
5.0.__add__
```

In [ ]:

```
callable(5.0.__add__)
```

In [ ]:

```
5.0.__add__()
```

In [ ]:

```
5.0.__add__(4)
```

In [ ]:

```
4.__add__
```

In [ ]:

```
(4).__add__
```

In [ ]:

```
(4).__add__(5)
```

In [ ]:

```
import sys
size = sys.getsizeof
print('Size of w is', size('w'), 'bytes.')
```

In [ ]:

```
size('walk')
```

In [ ]:

```
size(2)
```

In [ ]:

```
size(2**30 - 1)
```

In [ ]:

```
size(2**30)
```

In [ ]:

```
size(2**60-1)
```

In [ ]:

```
size(2**60)
```

In [ ]:

```
size(2**1000)
```

2 Names

2.1 Back to the Basics: Names

Every object has (zero or) one or more *names*, in one or more namespaces.
Understanding names is foundational to understanding Python and using it effectively

Names refer to objects. Namespaces are like dictionaries.

In [ ]:

```
dir()
```

IPython adds a lot of names to the global namespace! Let's workaround that.

In [ ]:

```
%%writefile dirp.py
def _dir(obj='__secret', _CLUTTER=dir()):
    """
    A version of dir that excludes clutter and private names.
    """
    if obj == '__secret':
        names = globals().keys()
    else:
        names = dir(obj)
    return [n for n in names if n not in _CLUTTER and not n.startswith('_')]

def _dirn(_CLUTTER=dir()):
    """
    Display the current global namespace, ignoring old names.
    """
    return dict([
        (n, v) for (n, v) in globals().items()
        if not n in _CLUTTER and not n.startswith('_')])
```

In [ ]:

```
%load dirp
```

In [ ]:

```
_dirn()
```

In [ ]:

```
a
```

Tries to find a in the namespace and it fails, so it returns nothing.

In [ ]:

```
a = 300
```

In [ ]:

```
_dirn()
```

This now prints a dictionary of {'a':300} because a is now in the namespace

In [ ]:

```
a
```

Python has *variables* in the mathematical sense - names that can vary, but not in the sense of boxes that hold values like you may be thinking about them. Imagine instead names or labels that you can add to an object or move to another object.

In [ ]:

```
a = 400
```

Simple name assignment and re-assignment are not operations on objects, they are namespace operations!

In [ ]:

```
_dirn()
```

In [ ]:

```
b = a
```

This is a statement so there's no return

In [ ]:

```
b
```

In [ ]:

```
a
```

In [ ]:

```
_dirn()
```

This returns a dictionary of a & b because we added both to the namespace

In [ ]:

```
id(a), id(b)
```

In [ ]:

```
id(a) == id(b)
```

This checks for equality

In [ ]:

```
a is b
```

This checks if both is the same object, not for equality

In [ ]:

```
del a
```

This removes a from the namespace dictionary -- it doesn't necessarily delete it persay

In [ ]:

```
_dirn()
```

In [ ]:

```
a
```

The `del` statement on a name is a namespace operation, i.e. it does not delete the object. Python will delete objects when they have no more names (when their reference count drops to zero).

Of course, given that the name `b` is just a name for an object and it's objects that have types, not names, there's no restriction on the type of object that the name `b` refers to.

In [ ]:

```
b = 'walk'
```

In [ ]:

```
b
```

In [ ]:

```
id(b)
```

In [ ]:

```
del b
```

In [ ]:

```
_dirn()
```

Object attributes are also like dictionaries, and "in a sense the set of attributes of an object also form a namespace." (https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces (https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces))

In [ ]:

```
class SimpleNamespace:
    pass
```

`SimpleNamespace` was added to the `types` module in Python 3.3

In [ ]:

```
import sys
if (sys.version_info.major, sys.version_info.minor) >= (3, 3):
    from types import SimpleNamespace
```

In [ ]:

```
p = SimpleNamespace()
```

SimpleNamespace is now a type that is assigned to p

In [ ]:

```
p
```

In [ ]:

```
p.__dict__
```

In [ ]:

```
p.x, p.y = 1.0, 2.0
```

In [ ]:

```
p.__dict__
```

In [ ]:

```
p.x, p.y
```

In [ ]:

```
i = 10
j = 10
i is j
```

i and j are the same object

In [ ]:

```
i == j
```

i and j are set to the same value

In [ ]:

```
i = 500
j = 500
i is j
```

Don't use is, use == to check for equality.

In [ ]:

```
i == j
```

Use == to check for equality. Only use `is` if you want to check identity, i.e. if two object references or names refer to the same object.

The reason == and `is` don't always match with `int` as shown above is that CPython pre-creates some frequently used `int` objects to increase performance. Which ones are documented in the source code, or we can figure out which ones by looking at their `id`s.

In [ ]:

```
import itertools
for i in itertools.chain(range(-7, -3), range(254, 259)):
    print(i, id(i))
```

Python pre-creates integers from -5 to 258 in its namespace

2.2 Exercises: Names

In [ ]:

```
dir()
```

In [ ]:

```
_dir = dir
```

If `dir()` returns too many names define and use _dir instead. Or use `dirp.py` from above. If you're running Python without the IPython notebook plain old `dir` should be fine.

In [ ]:

```
def _dir(_CLUTTER=dir()):
    """
    Display the current global namespace, ignoring old names.
    """
    return [n for n in globals()
            if n not in _CLUTTER and not n.startswith('_')]
```

In [ ]:

```
v = 1
```

In [ ]:

```
v
```

In [ ]:

```
_dir()
```

In [ ]:

```
type(v)
```

In [ ]:

```
w = v
```

In [ ]:

```
v is w
```

In [ ]:

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

In [ ]:

```
n = m
n
```

In [ ]:

```
_dir()
```

In [ ]:

```
m is n
```

In [ ]:

```
m[1] = 'two'
m, n
```

Because n is m, if you change m, n will change too

In [ ]:

```
int.__add__
```

In [ ]:

```
int.__add__ = int.__sub__
```

In [ ]:

```
from sys import getrefcount as refs
```

In [ ]:

```
refs(None)
```

In [ ]:

```
refs(object)
```

In [ ]:

```
sentinel_value = object()
```

This creates a new object

In [ ]:

```
refs(sentinel_value)
```

This will say there are 2 references. 1 is the name of sentinel_value in the namespace & the second is the name in the refs function because when refs was running, there was 2!

Use `object()` to create a unique object which is not equal to any other object, for example to use as a sentinel value.

In [ ]:

```
sentinel_value == object()
```

In [ ]:

```
sentinel_value == sentinel_value
```

In [ ]:

```
refs(1)
```

In [ ]:

```
refs(2)
```

In [ ]:

```
refs(25)
```

In [ ]:

```
[(i, refs(i)) for i in range(100)]
```

In [ ]:

```
i, j = 1, 2
```

In [ ]:

```
i, j
```

In [ ]:

```
i, j = j, i
```

In [ ]:

```
i, j
```

In [ ]:

```
i, j, k = (1, 2, 3)
```

In [ ]:

```
i, j, k = 1, 2, 3
```

In [ ]:

```
i, j, k = [1, 2, 3]
```

In [ ]:

```
i, j, k = 'ijk'
```

Extended iterable unpacking is available in Python 3:

In [ ]:

```
i, j, k, *rest = 'ijklmnop'
```
i=i, j=j, k=k, rest = [l, m, n, o, p]

In [ ]:

```
i, j, k, rest
```

In [ ]:

```
first, *middle, second_last, last = 'abcdefg'
```
first = a, second_last = f, last = g

In [ ]:

```
first, middle, second_last, last
```

In [ ]:

```
i, *middle, j = 'ij'
```

In [ ]:

```
i, middle, j
```

3 More About Namespaces

3.1 Namespace Scopes and Search Order

Review:

- A *namespace* is a mapping from valid identifier names to objects. Think of it as a dictionary.
- Simple assignment (=) and `del` are namespace operations, not operations on objects.

Terminology and Definitions:

- A *scope* is a section of Python code where a namespace is *directly* accessible.
- For an *indirectly* accessible namespace you access values via dot notation, e.g. `p.x` or `sys.version_info.major`.
- The (*direct*) namespace search order is (from [http://docs.python.org/3/tutorial (http://docs.python.org/3/tutorial)](http://docs.python.org/3/tutorial)):
  - The innermost scope contains local names
  - The namespaces of enclosing functions, searched starting with the nearest enclosing scope; (or the module if outside any function)
  - The middle scope contains the current module's global names
  - 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):
  - *name* = i.e. assignment
  - `del` *name*
  - `import` *name*
  - `def` *name*
  - `class` *name*
  - function parameters: `def foo(`*name*`):`
  - `for` loop: `for` *name* `in ...`
  - except clause: `Exception as` *name*`:`
  - with clause: `with open(filename) as` *name*`:`
  - docstrings: `__doc__`

You should never reassign built-in names..., but let's do so to explore how name scopes work.

In [ ]:

```
len
```

In [ ]:

```
def f1():
    print('In f1() a line 1, len = {}'.format(len))
    def len():
        len = range(3)
        print("In f1's local len(), len is {}".format(len))
        return len
    print('In f1() at line 6, len = {}'.format(len))
    result = len()
    print('Returning result: {!r}'.format(result))
    return result
```

In [ ]:

```
f1()
```

In [ ]:

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

In [ ]:

```
f2()
```

In [ ]:

```
len
```

In [ ]:

```
len = 99
```

We are overwriting a function here, which works, but it's a stupid idea

In [ ]:

```
len
```

In [ ]:

```
def print_len(s):
    print('len(s) == {}'.format(len(s)))
```

In [ ]:

```
print_len('walk')
```

This errors since len=4 and an int object isn't callable

In [ ]:

```
len
```

In [ ]:

```
del len
```

In [ ]:

```
len
```

This brings back the function since del deletes len from the namespace, but the function is still here

In [ ]:

```
print_len('walk')
```

In [ ]:

```
pass
```

In [ ]:

```
pass = 3
```

This doesn't work because you can't overwrite a keyword in Python.

Keywords at https://docs.python.org/3/reference/lexical_analysis.html#keywords (https://docs.python.org/3/reference/lexical_analysis.html#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
```

3.2 Namespaces: Function Locals

Let's look at some surprising behaviour.

In [ ]:

```
x = 1
def test_outer_scope():
    print('In test_outer_scope x ==', x)
```

In [ ]:

```
test_outer_scope()
```

In [ ]:

```
def test_local():
    x = 2
    print('In test_local x ==', x)
```

In [ ]:

```
x
```

x is still 1 because we define x inside of a function, which of course, doesn't make it's way 'upwards'

In [ ]:

```
test_local()
```

In [ ]:

```
x
```

In [ ]:

```
def test_unbound_local():
    print('In test_unbound_local  ==', x)
    x = 3
```

In [ ]:

```
x
```

In [ ]:

```
test_unbound_local()
```

This produces an error. We do a namespace change in last function definition but it's before we set x.

In [ ]:

```
x
```

Let's introspect the function `test_unbound_local` to help us understand this error.

In [ ]:

```
test_unbound_local.__code__
```

In [ ]: How many arguments are passed?

```
test_unbound_local.__code__.co_argcount  # count of positional args
```

In [ ]:

```
test_unbound_local.__code__.co_name  # function name
```

In [ ]:

```
test_unbound_local.__code__.co_names  # names used in bytecode
```

In [ ]:

```
test_unbound_local.__code__.co_nlocals  # number of locals
```
variables

In [ ]:

```
test_unbound_local.__code__.co_varnames  # names of locals
```
variables

See "Code objects" at https://docs.python.org/3/reference/datamodel.html?highlight=co_nlocals#the-standard-type-hierarchy (https://docs.python.org/3/reference/datamodel.html?highlight=co_nlocals#the-standard-type-hierarchy)

In [ ]:

```
import dis
```

In [ ]:

```
dis.dis(test_unbound_local.__code__.co_code)
```

The use of `x` by LOAD_FAST happens before it's set by STORE_FAST.

> "This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to x, the compiler recognizes it as a local variable. Consequently when the earlier print x attempts to print the uninitialized local variable and an error results." -- https://docs.python.org/3/faq/programming.html#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value (https://docs.python.org/3/faq/programming.html#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value)

To explore this further on your own compare these two:

```
dis.dis(codeop.compile_command('def t1(): a = b; b = 7'))
dis.dis(codeop.compile_command('def t2(): b = 7; a = b'))
```

In [ ]:

```
def test_global():
    global x
    print('In test_global before, x ==', x)
    x = 4
    print('In test_global after, x ==', x)
```

Remember that x=1 previously so it carries here.
But now that x=4 and x is global, x will become 4.

In [ ]:

```
x
```

In [ ]:

```
test_global()
```

In [ ]:

```
x
```

In [ ]:

```
test_global.__code__.co_varnames
```

In [ ]:

```
def test_nonlocal():
    x = 5
    def test6():
        nonlocal x
        print('test6 before x ==', x)
        x = 6
        print('test6 after x ==', x)
    print('test_nonlocal before x ==', x)
    test6()
    print('test_nonlocal after x ==', x)
```

In [ ]:

```
x = 1
```

In [ ]:

```
x
```

In [ ]:

```
test_nonlocal()
```

In [ ]:

```
x
```

5 Functions

5.4 Function Arguments are Passed by Assignment

Can functions modify the arguments passed in to them?

When a caller passes an argument to a function, the function starts execution with a local name (the parameter from its signature) referring to the argument object passed in.

In [ ]:

```
def test_1a(s):
    print('Before:', s)
    s += ' two'
    print('After:', s)
```

In [ ]:

```
s1 = 'one'
s1
```

In [ ]:

```
test_1a(s1)
```

In [ ]:

```
s1
```

To see more clearly why `s1` is still a name for 'one', consider this version which is functionally equivalent but has two changes highlighted in the comments:

In [ ]:

```
def test_1b(s):
    print('Before:', s)
    s = s + ' two'   # Changed from +=
    print('After:', s)
```

In [ ]:

```
test_1b('one')  # Changed from s1 to 'one'
```

In both cases the name `s` at the beginning of `test_1a` and `test_1b` was a name that referred to the `str` object `'one'`, and in both the function-local name `s` was reassigned to refer to the new `str` object `'hello there'`.

Let's try this with a `list`.

In [ ]:

```
def test_2a(m):
    print('Before:', m)
    m += [4]  # list += list is shorthand for list.extend(list)
    print('After:', m)
```

In [ ]:

```
m1 = [1, 2, 3]
```

In [ ]:

```
m1
```

In [ ]:

```
test_2a(m1)
```

This finds m1 in the namespace and passes it by reference to test_2a.

In [ ]:

```
m1
```

7 How Classes Work

7.1 Deconstructing the Class Statement

- The `class` statement starts a block of code and creates a new namespace. All namespace changes in the block, e.g. assignment and function definitions, are made in that new namespace. Finally it adds the class name to the namespace where the class statement appears.
- Instances of a class are created by calling the class: `ClassName()` or `ClassName(args)`.
- `ClassName.__init__(<new object>, ...)` is called automatically, and is passed the instance of the class already created by a call to the `__new__` method.
- 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 argument to the method.

Classes create a new namespace and all changes made in the class are made to that namespace.

In [ ]:

```
class Number:  # In Python 2.x use "class Number(object):"
    """A number class."""
    __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
```

In [ ]:

```
Number
```

Number is now a new name in the namespace

In [ ]:

```
Number.__version__
```

In [ ]:

```
Number.__doc__
```

In [ ]:

```
help(Number)
```

In [ ]:

```
Number.__init__
```

Calls def inside of this class

In [ ]:

```
Number.add
```

Calls def inside of this class

In [ ]:

```
dir(Number)
```

In [ ]:

```
def dir_public(obj):
    return [n for n in dir(obj) if not n.startswith('__')]
```

In [ ]:

```
dir_public(Number)
```

Returns just 'add' because add is the only function we called thusfar that didn't begin __

In [ ]:

```
number2 = Number(2)
```

In [ ]:

```
number2.amount
```

In [ ]:

```
number2
```

In [ ]:

```
number2.__init__
```

In [ ]:

```
number2.add
```

In [ ]:

```
dir_public(number2)
```

Now this prints amount because we called amount, even though amount wasn't in the class

In [ ]:

```
set(dir(number2)) ^ set(dir(Number))  # symmetric_difference
```

Prints {'amount'}

In [ ]:

```
number2.__dict__
```

In [ ]:

```
Number.__dict__
```

In [ ]:

```
'add' in Number.__dict__
```

In [ ]:

```
number2.add
```

In [ ]:

```
number2.add(3)
```

Here's some unusual code ahead which will help us think carefully about how Python works.

In [ ]:

```
Number.add
```

In [ ]:

```
def add(self, value):  # Earlier definition
        return self.amount + value
```

In [ ]:

```
Number.add(2)
```

In [ ]:

```
Number.add(2, 3)
```

This fails because 2 -- the first attribute -- does not have an 'amount' attribute.

In [ ]: number2 = 5 from above

```
Number.add(number2, 3)
```
This works because number2 -- the first parameter -- has an amount

In [ ]:

```
number2.add(3)
```

In [ ]:

```
def __init__(self, amount):   # Earlier definition
    self.amount = amount
```

In [ ]:

```
Number.__init__
```

In [ ]:

```
help(Number.__init__)
```

Here's some code that's downright risky, but instructive. You should never need to do this in your code.

In [ ]:

```
def set_double_amount(number, amount):
    number.amount = 2 * amount
```

In [ ]:

```
Number.__init__ = set_double_amount
```
This works -- number has an amount attr. Don't do this though.

In [ ]:

```
Number.__init__
```
This is now a function of __main__.set_double_amount

In [ ]:

```
help(Number.__init__)
```

In [ ]:

```
number4 = Number(2)
```

In [ ]:

```
number4.amount
```

In [ ]:

```
number4.add(5)
```

In [ ]:

```
number4.__init__
```

In [ ]:

```
number2.__init__
```

In [ ]:

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

Let's add a `mul` method. However, I will intentionally make a mistake.

In [ ]:

```
number4.mul = multiply_by
```

In [ ]:

```
number4.mul
```

In [ ]:

```
number4.mul(5)
```

This doesn't work because it's not a bound method, it's a function. We need to add it as an arg.
This is trying to find the attribute mul inside of number4.

In [ ]:

```
number4.mul(number4, 5)
```

Where's the mistake?

In [ ]:

```
number10 = Number(5)
```

In [ ]:

```
number10.mul
```

In [ ]:

```
dir_public(number10)
```

In [ ]:

```
dir_public(Number)
```

In [ ]:

```
dir_public(number4)
```

In [ ]:

```
Number.mul = multiply_by
```

In [ ]:

```
number10.mul(5)
```

This works because mul is now a valid attribute for Number.

In [ ]:

```
number4.mul(5)
```

This doesn't work because number4 has mul in its namespace as an attribute too so we need to delete it so it uses the last mul attribute definition.

In [ ]:

```
dir_public(number4)
```

In [ ]:

```
number4.__dict__
```

In [ ]:

```
del number4.mul
```

In [ ]:

```
number4.__dict__
```

In [ ]:

```
dir_public(number4)
```

In [ ]:

```
number4.mul
```

In [ ]:

```
Number.mul
```

In [ ]:

```
number4.mul(5)
```

Let's look behind the curtain to see how class instances work in Python.

In [ ]:

```
Number
```

In [ ]:

```
number4
```

In [ ]:

```
Number.add
```

This is a function because we're calling it off a class

In [ ]:

```
number4.add
```

This is a bound method because we're calling it off a variable/name

Bound methods are handy.

In [ ]:

```
add_to_4 = number4.add
```

In [ ]:

```
add_to_4(6)
```

In [ ]:

```
dir_public(number4)
```

In [ ]:

```
dir(number4.add)
```

In [ ]:

```
dir_public(number4.add)
```

In [ ]:

```
set(dir(number4.add)) - set(dir(Number.add))
```

The bound method has everything the function has but it also has __func__ and __self__

In [ ]:

```
number4.add.__self__
```

In [ ]:

```
number4.add.__self__ is number4
```

In [ ]:

```
number4.add.__func__
```

In [ ]:

```
number4.add.__func__ is Number.add
```

In [ ]:

```
number4.add.__func__ is number10.add.__func__
```

In [ ]:

```
number4.add(5)
```

So here's approximately how Python executes `number4.add(5)`:

In [ ]:

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

7.2 Creating Classes with the type Function

"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: Demystifying Deferreds, Decorators, and Declarations* at PyCon 2010

`type(name, bases, dict)` is the default function that gets called when Python read a `class` statement.

In [ ]:

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

Let's use the type function to build a class.

In [ ]:

```
def init(self, amount):
    self.amount = amount
```

In [ ]:

```
def add(self, value):
    """Add a value to the number."""
    print('Call: add({!r}, {})'.format(self, value))
    return self.amount + value
```

In [ ]:

```
Number = type(
    'Number', (object,),
    dict(__version__='1.0', __init__=init, add=add))
```

This is a metaclass

In [ ]:

```
number3 = Number(3)
```

In [ ]:

```
type(number3)
```

In [ ]:

```
number3.__class__
```

In [ ]:

```
number3.__dict__
```

In [ ]:

```
number3.amount
```

In [ ]:

```
number3.add(4)
```

Remember, here's the normal way to create a class:

In [ ]:

```
class Number:
    __version__='1.0'

    def __init__(self, amount):
        self.amount = amount

    def add(self, value):
        return self.amount + value
```

We can customize how classes get created.
https://docs.python.org/3/reference/datamodel.html#customizing-class-creation
(https://docs.python.org/3/reference/datamodel.html#customizing-class-creation)

> 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.

The following makes explicit that the `metaclass`, i.e. the callable that Python should use to create a class, is the built-in function `type`.

In [ ]:

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

7.3 Exercises: The Class Statement

Test your understanding of the mechanics of class creation with some very unconventional uses of those mechanics.

What does the following code do? Note that `return_5` ignores arguments passed to it.

In [ ]:

```
def return_5(name, bases, namespace):
    print('Called return_5({!r})'.format((name, bases, namespace)))
    return 5
```

In [ ]:

```
return_5(None, None, None)
```

In [ ]:

```
x = return_5(None, None, None)
```

In [ ]:

```
x
```

In [ ]:

```
type(x)
```

The syntax for specifying a metaclass changed in Python 3 so choose appropriately.

In [ ]:

```
class y(object):  # Python 2.x
    __metaclass__ = return_5
```

In [ ]:

```
class y(metaclass=return_5):  # Python 3.x
        pass
```

In [ ]:

```
y
```

In [ ]:

```
type(y)
```

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

In [ ]:

```
def return_6(klass):
    print('Called return_6({!r})'.format(klass))
    return 6
```

In [ ]:

```
return_6(None)
```

In [ ]:

```
@return_6
class z:
    pass
```

In [ ]:

```
z
```

In [ ]:

```
type(z)
```

9 Iterators and Generators

9.1 Iterables, Iterators, and the Iterator Protocol

- 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.

In [ ]:

```
for i in 'abc':
    print(i)
```

In [ ]:

```
iterator = iter('ab')
```

In [ ]:

```
iterator.__next__()
```

In [ ]:

```
iterator.__next__()
```

In [ ]:

```
iterator.__next__()
```

In [ ]:

```
iterator.__next__()
```

In [ ]:

```
iterator = iter('ab')
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

`next()` just calls `__next__()`, but you can pass it a second argument:

In [ ]:

```
iterator = iter('ab')
```

In [ ]:

```
next(iterator, 'z')
```

In [ ]:

```
next(iterator, 'z')
```

In [ ]:

```
next(iterator, 'z')
```

In [ ]:

```
next(iterator, 'z')
```

- `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`.

In [ ]:

```
class MyList:
    """Demonstrate the iterator protocol"""
    def __init__(self, sequence):
        self.items = sequence

    def __getitem__(self, key):
        print('called __getitem__({})'
              .format(key))
        return self.items[key]
```

In [ ]:

```
m = MyList('ab')
```

In [ ]:

```
m.__getitem__(0)
```

In [ ]:

```
m.__getitem__(1)
```

In [ ]:

```
m.__getitem__(2)
```

In [ ]:

```
m[0]
```

In [ ]:

```
m[1]
```

In [ ]:

```
m[2]
```

In [ ]:

```
hasattr(m, '__iter__')
```

In [ ]:

```
hasattr(m, '__getitem__')
```

In [ ]:

```
iterator = iter(m)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
list(m)
```

In [ ]:

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

9.3 Generator Functions

In [ ]:

```
def list123():
    print('Before first yield')
    yield 1
    print('Between first and second yield')
    yield 2
    print('Between second and third yield')
    yield 3
    print('After third yield')
```

In [ ]:

```
list123
```

In [ ]:

```
list123()
```

In [ ]:

```
iterator = list123()
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
for i in list123():
    print(i)
```

In [ ]:

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

In [ ]:

```
iterator = even(3)
```

In [ ]:

```
iterator
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
next(iterator)
```

In [ ]:

```
for i in even(3):
    print(i)
```

In [ ]:

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

Compare these versions

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

```
%%writefile eg.txt
This is some sample
text.  It has a couple
of paragraphs.

Each paragraph has at
least one sentence.

Most paragraphs have
two.
```

In [ ]:

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

In [ ]:

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

10 Taking Advantage of First Class Objects

10.1 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 the only reasonable solution would 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.

Let's iteratively write code to get this behaviour:

```
assert calc('7+3') == 10
assert calc('9-5') == 4
assert calc('9/3') == 3
```

In [ ]:

```
7+3
```

In [ ]:

```
expr = '7+3'
```

In [ ]:

```
lhs, op, rhs = expr
```

In [ ]:

```
lhs, op, rhs
```

In [ ]:

```
lhs, rhs = int(lhs), int(rhs)
```

In [ ]:

```
lhs, op, rhs
```

In [ ]:

```
op, lhs, rhs
```

In [ ]:

```
def perform_operation(op, lhs, rhs):
    if op == '+':
        return lhs + rhs
    if op == '-':
        return lhs - rhs
    if op == '/':
        return lhs / rhs
```

In [ ]:

```
perform_operation('+', 7, 3) == 10
```

The `perform_operation` function has a lot of boilerplate repetition. Let's use a data structure instead to use less code and make it easier to extend.

In [ ]:

```
import operator
```

In [ ]:

```
operator.add(7, 3)
```

In [ ]:

```
OPERATOR_MAPPING = {
    '+': operator.add,
    '-': operator.sub,
    '/': operator.truediv,
    }
```

In [ ]:

```
OPERATOR_MAPPING['+']
```

In [ ]:

```
OPERATOR_MAPPING['+'](7, 3)
```

In [ ]:

```
def perform_operation(op, lhs, rhs):
    return OPERATOR_MAPPING[op](lhs, rhs)
```

In [ ]:

```
perform_operation('+', 7, 3) == 10
```

In [ ]:

```
def calc(expr):
    lhs, op, rhs = expr
    lhs, rhs = int(lhs), int(rhs)
    return perform_operation(op, lhs, rhs)
```

In [ ]:

```
calc('7+3')
```

In [ ]:

```
calc('9-5')
```

In [ ]:

```
calc('9/3')
```

In [ ]:

```
calc('3*4')
```

In [ ]:

```
OPERATOR_MAPPING['*'] = operator.mul
```

In [ ]:

```
calc('3*4')
```

Let's look at another example. Suppose we have data where every line is fixed length with fixed length records in it and we want to pull fields out of it by name:

```
PYTHON_RELEASES = [
    'Python 3.4.0 2014-03-17',
    'Python 3.3.0 2012-09-29',
    'Python 3.2.0 2011-02-20',
    'Python 3.1.0 2009-06-26',
    'Python 3.0.0 2008-12-03',
    'Python 2.7.9 2014-12-10',
    'Python 2.7.8 2014-07-02',
]

release34 = PYTHON_RELEASES[0]

release = ReleaseFields(release34)  # 3.4.0
assert release.name == 'Python'
assert release.version == '3.4.0'
assert release.date == '2014-03-17'
```

This works:

In [ ]:

```
class ReleaseFields:
    def __init__(self, data):
        self.data = data

    @property
    def name(self):
        return self.data[0:6]

    @property
    def version(self):
        return self.data[7:12]

    @property
    def date(self):
        return self.data[13:23]
```

In [ ]:

```
release34 = 'Python 3.4.0 2014-03-17'
```

In [ ]:

```
release = ReleaseFields(release34)
```

In [ ]:

```
assert release.name == 'Python'
assert release.version == '3.4.0'
assert release.date == '2014-03-17'
```

However, the following is better especially if there are many fields or as part of a libary which handle lots of different record formats:

In [ ]:

```
class ReleaseFields:
    slices = {
        'name': slice(0, 6),
        'version': slice(7, 12),
        'date': slice(13, 23)
        }

    def __init__(self, data):
        self.data = data

    def __getattr__(self, attribute):
        if attribute in self.slices:
            return self.data[self.slices[attribute]]
        raise AttributeError(
            "{!r} has no attribute {!r}"
            .format(self, attribute))
```

In [ ]:

```
release = ReleaseFields(release34)
```

In [ ]:

```
assert release.name == 'Python'
assert release.version == '3.4.0'
assert release.date == '2014-03-17'
```

Confirm that trying to access an attribute that doesn't exist fails correctly. (Note they won't in Python 2.x unless you add `(object)` after `class ReleaseFields`).

In [ ]:

```
release.foo == 'exception'
```

If you find yourself writing lots of boilerplate code as in the first versions of the calculator and fixed length record class above, you may want to try changing it to use a Python data structure with first class objects.

10.2 Binding Data with Functions

It is often useful to bind data to a function. A method clearly does that, binding the instance's attributes with the method behaviour, but it's not the only way.

In [ ]:

```
def log(severity, message):
    print('{}: {}'.format(severity.upper(), message))
```

In [ ]:

```
log('warning', 'this is a warning')
```

In [ ]:

```
log('error', 'this is an error')
```

Create a new function that specifies one argument.

In [ ]:

```
def warning(message):
    log('warning', message)
```

In [ ]:

```
warning('this is a warning')
```

Create a closure from a function that specifies an argument.

In [ ]:

```
def create_logger(severity):
    def logger(message):
        log(severity, message)
    return logger
```

In [ ]:

```
warning2 = create_logger('warning')
```

In [ ]:

```
warning2('this is a warning')
```

Create a partial function.

In [ ]:

```
import functools
```

In [ ]:

```
warning3 = functools.partial(log, 'warning')
```

In [ ]:

```
warning3
```

In [ ]:

```
warning3.func is log
```

In [ ]:

```
warning3.args, warning3.keywords
```

In [ ]:

```
warning3('this is a warning')
```

Use a bound method.

In [ ]:

```
SENTENCE_PUNCUATION = '.?!'
```

In [ ]:

```
sentence = 'This is a sentence!'
```

In [ ]:

```
sentence[-1] in SENTENCE_PUNCUATION
```

In [ ]:

```
'.' in SENTENCE_PUNCUATION
```

In [ ]:

```
SENTENCE_PUNCUATION.__contains__('.')
```

In [ ]:

```
SENTENCE_PUNCUATION.__contains__(',')
```

In [ ]:

```
is_end_of_a_sentence = SENTENCE_PUNCUATION.__contains__
```

In [ ]:

```
is_end_of_a_sentence('.')
```

In [ ]:

```
is_end_of_a_sentence(',')
```

Create a class with a `__call__` method.

In [ ]:

```
class SentenceEndsWith:
    def __init__(self, characters):
        self.punctuation = characters

    def __call__(self, sentence):
        return sentence[-1] in self.punctuation
```

In [ ]:

```
is_end_of_a_sentence_dot1 = SentenceEndsWith('.')
```

In [ ]:

```
is_end_of_a_sentence_dot1('This is a test.')
```

In [ ]:

```
is_end_of_a_sentence_dot1('This is a test!')
```

In [ ]:

```
is_end_of_a_sentence_any = SentenceEndsWith('.!?')
```

In [ ]:

```
is_end_of_a_sentence_any('This is a test.')
```

In [ ]:

```
is_end_of_a_sentence_any('This is a test!')
```

Another way that mutable data can be bound to a function is with parameter evaluation, which is sometimes done by mistake.

In [ ]:

```
def f1(parameter=print('The parameter is initialized now!')):
    if parameter is None:
        print('The parameter is None')
    return parameter
```

In [ ]:

```
f1()
```

In [ ]:

```
f1() is None
```

In [ ]:

```
f1('Not None')
```

In [ ]:

```
def f2(parameter=[0]):
    parameter[0] += 1
    return parameter[0]
```

In [ ]:

```
f2()
```

In [ ]:

```
f2()
```

In [ ]:

```
f2()
```

In [ ]:

```
f2()
```