Introduction to Python Epiphanies: Exploring Fundamental Concepts

Introduction

- The target audience is intermediate Python users looking for a deeper understanding of the language. It attempts to correct some common misperceptions of how Python works. While similar to many other programming languages, Python is quite different from some in subtle and important ways.
- Almost all of the material in the video is presented in the interactive Python prompt (aka the Read Eval Print Loop or REPL). I'll be using an IPython notebook but you can use Python without IPython just fine.
- I'm using Python 3.4 and I suggest you do the same unless you're familiar with the differences between Python versions 2 and 3 and prefer to use Python 2.x.
- There are some intentional code errors in both the regular presentation material and the exercises. The purpose of the intentional errors is to foster learning from how things fail.

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 [ ]:
```python
b'a bytes literal'
```

In [ ]:
```python
(1, 2)
```

In [ ]:
```python
[1, 2]
```

In [ ]:
```python
{'one': 1, 'two': 2}
```

In [ ]:
```python
{'one', 'two'}
```

Some constants are created on startup and have names.

In [ ]:
```python
False, True
```

In [ ]:
```python
None
```

In [ ]:
```python
NotImplemented, Ellipsis
```

There are also some built-in types and functions.

In [ ]:
```python
int, list
```

In [ ]:
```python
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.star
tswith('_')]

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
```

In [ ]:
```
a = 300
```

In [ ]:
```
_dirn()
```

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
```

In [ ]:
```
b
```

In [ ]:
```
a
```

In [ ]:
```
_dirn()
```

In [ ]:
```
id(a), id(b)
```

Erratum: There is an error in the video where I talk about the next line as though it were checking for equality between a and b instead of checking that their ids are equal.

In [ ]:
```
id(a) == id(b)
```

In [ ]:
```
a is b
```

In [ ]:
```
del a
```

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()
```

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
```

In [ ]:
```
i == j
```

In [ ]:
```
i = 500
j = 500
i is j
```

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))
```

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
```

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()
```

In [ ]:
```
refs(sentinel_value)
```

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 only in Python 3:

In [ ]:
```
i, j, k, *rest = 'ijklmnop'
```

In [ ]:
```
i, j, k, rest
```

In [ ]:
```
first, *middle, second_last, last = 'abcdefg'
```

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)):
  - 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 [ ]:
```python
def f1():
    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 [ ]:
```python
f1()
```

In [ ]:
```python
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 [ ]:
```python
f2()
```

In [ ]:
```python
len
```

In [ ]:
```python
len = 99
```

In [ ]:
```python
len
```

In [ ]:
```python
def print_len(s):
    print('len(s) == {}'.format(len(s)))
```

In [ ]:
```python
print_len('walk')
```

In [ ]:
```python
len
```

In [ ]:
```python
del len
```

In [ ]:
```python
len
```

In [ ]:
```python
print_len('walk')
```

In [ ]:
```
pass
```

In [ ]:
```
pass = 3
```

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
```

In [ ]:
```
test_local()
```

In [ ]:
```
x
```

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

In [ ]:
```
x
```

In [ ]:
```
test_unbound_local()
```

In [ ]:
```
x
```

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

In [ ]:
```
test_unbound_local.__code__
```

In [ ]:
```
test_unbound_local.__code__.co_argcount  # count of positional a
rgs
```

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
```

In [ ]:
```
test_unbound_local.__code__.co_varnames  # names of locals
```

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

In [ ]:
```python
x
```

In [ ]:
```python
test_global()
```

In [ ]:
```python
x
```

In [ ]:
```python
test_global.__code__.co_varnames
```

In [ ]:
```python
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 [ ]:
```python
x = 1
```

In [ ]:
```python
x
```

In [ ]:
```python
test_nonlocal()
```

In [ ]:
```
x
```

3.3 The Built-ins Namespace

Restart Python to unclutter the namespace.

In [ ]:
```
%%javascript
IPython.notebook.kernel.restart();
```

In [ ]:
```
[n for n in dir() if not n.startswith('_')]
```

There are lots of built-in names that `dir()` doesn't show us. Let's use some Python to explore all the builtin names by category.

In [ ]:
```
import builtins, collections, inspect, textwrap
fill = textwrap.TextWrapper(width=60).fill
def pfill(pairs):
    """Sort and print first of every pair"""
    print(fill(' '.join(list(zip(*sorted(pairs)))[0])))
```

Collect all members of `builtins`:

In [ ]:
```
members = set([
    m for m in inspect.getmembers(builtins)
    if not m[0].startswith('_')])
len(members)
```

Pull out just the `exceptions`:

In [ ]:
```
exceptions = [
    (name, obj) for (name, obj) in members
    if inspect.isclass(obj) and
    issubclass(obj, BaseException)]
members -= set(exceptions)
len(exceptions), len(members)
```

In [ ]:
```
pfill(exceptions)
```

[https://docs.python.org/3/library/exceptions.html#exception-hierarchy (https://docs.python.org/3/library/exceptions.html#exception-hierarchy)](https://docs.python.org/3/library/exceptions.html#exception-hierarchy):

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
```

```
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

In [ ]:
```
pfill(members)
```

Most are one of these two types:

In [ ]:
```
type(int), type(len)
```

Print them:

In [ ]:
```
bnames = collections.defaultdict(set)
for name, obj in members:
    bnames[type(obj)].add((name, obj))
for typ in [type(int), type(len)]:
    pairs = bnames.pop(typ)
    print(typ)
    pfill(pairs)
    print()
```

The leftovers:

In [ ]:
```
for typ, pairs in bnames.items():
    print('{}: {}'.format(typ, ' '.join((n for (n, o) in pair
s))))
```

3.4 Exercises: The Built-ins Namespace

In [ ]:
```
[k for k in locals().keys() if not k.startswith('_')]
```

In [ ]:
```
[k for k in globals().keys() if not k.startswith('_')]
```

In the REPL these are the same:

In [ ]:
```
locals() == globals()
```

The following code is *not* recommended but it reminds us that namespaces are like dictionaries.

In [ ]:
```
x = 0
```

In [ ]:
```
x
```

In [ ]:
```
locals()['x']
```

In [ ]:
```
locals()['x'] = 1
```

In [ ]:
```
locals()['x']
```

In [ ]:
```
x
```

If you're tempted to use it, try this code which due to "fast locals" doesn't do what you might expect:

In [ ]:
```
def f():
    locals()['x'] = 5
    print(x)
```

4 Import

4.1 The import Statement

Remember, these change or modify a namespace:

- Simple assignment (=) and `del`
- [`globals()` and `locals()`]
- `import`

- `def`
- `class`
- [Also function parameters, `for`, `except`, `with`, and docstrings.]

Next we'll explore `import`.

In [ ]:
```
%load dirp
```

In [ ]:
```
_dir()
```

In [ ]:
```
import pprint
_dir()
```

In [ ]:
```
pprint
```

In [ ]:
```
_dir(pprint)
```

In [ ]:
```
pprint.pformat
```

In [ ]:
```
pprint.pprint
```

In [ ]:
```
pprint.foo
```

In [ ]:
```
pprint.foo = 'Python is dangerous'
pprint.foo
```

In [ ]:
```
from pprint import pformat as pprint_pformat
_dir()
```

In [ ]:
```
pprint.pformat is pprint_pformat
```

In [ ]:
```
pprint
```

In [ ]:
```
pprint.pformat
```

In [ ]:
```
del pprint
import pprint as pprint_module
_dir()
```

In [ ]:
```
pprint_module.pformat is pprint_pformat
```

In [ ]:
```
math
```

In [ ]:
```
dir(math)
```

In [ ]:
```
del math
```

In [ ]:
```
import math
```

Why doesn't `import math` give a `NameError`?

In [ ]:
```
math
```

In [ ]:
```
del math
```

What if we don't know the name of the module until run-time?

In [ ]:
```
import importlib
```

In [ ]:
```
importlib.import_module('math')
```

In [ ]:
```
math_module = importlib.import_module('math')
```

In [ ]:
```
math_module.pi
```

In [ ]:
```
math
```

In [ ]:
```
module_name = 'math'
```

In [ ]:
```
import module_name
```

In [ ]:
```
import 'math'
```

In [ ]:
```
import math
```

4.2 Exercises: The import Statement

In [ ]:
```
import pprint
```

In [ ]:
```
dir(pprint)
```

In [ ]:
```
pprint.__doc__
```

In [ ]:
```
pprint.__file__
```

In [ ]:
```
pprint.__name__
```

In [ ]:
```
from pprint import *
```

In [ ]:
```
[n for n in dir() if not n.startswith('_')]
```

In [ ]:
```
import importlib
```

In [ ]:
```
help(importlib.reload)
```

In [ ]:
```
importlib.reload(csv)
```

In [ ]:
```
importlib.reload('csv')
```

In [ ]:
```
import csv
```

In [ ]:
```
importlib.reload('csv')
```

In [ ]:
```
importlib.reload(csv)
```

In [ ]:
```
import sys
```

In [ ]:
```
sys.path
```

5 Functions

5.1 Functions

In [ ]:
```
def f():
    pass
```

In [ ]:
```
f.__name__
```

In [ ]:
```
f
```

In [ ]:
```
f.__name__ = 'g'
```

In [ ]:
```
g
```

In [ ]:
```
f.__name__
```

In [ ]:
```
f
```

In [ ]:
```
f.__qualname__  # Only in Python >= 3.3
```

In [ ]:
```
f.__qualname__ = 'g'
f
```

In [ ]:
```
f.__dict__
```

In [ ]:
```
f.foo = 'bar'
f.__dict__
```

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

In [ ]:
```python
f.__defaults__
```

In [ ]:
```python
f(1, 2)
```

In [ ]:
```python
f(a=1, b=2)
```

In [ ]:
```python
f(b=1, a=2)
```

In [ ]:
```python
f(1, 2, 3)
```

In [ ]:
```python
f(1, 2, k2=4)
```

In [ ]:
```python
f(1, k1=3)  # Fails
```

In [ ]:
```python
f(1, 2, 3, 4, 5, 6)
```

In [ ]:
```python
f(1, 2, 3, 4, keya=7, keyb=8)
```

In [ ]:
```python
f(1, 2, 3, 4, 5, 6, keya=7, keyb=8)
```

In [ ]:
```python
f(1, 2, 3, 4, 5, 6, keya=7, keyb=8, 9)
```

In [ ]:
```python
def g(a, b, *args, c=None):
    print('a: {!r}, b: {!r}, '
        'args: {!r}, c: {!r}'
        .format(a, b, args, c))
```

In [ ]:
```python
g.__defaults__
```

In [ ]:
```python
g.__kwdefaults__
```

In [ ]:
```python
g(1, 2, 3, 4)
```

In [ ]:
```python
g(1, 2, 3, 4, c=True)
```

Keyword-only arguments in Python 3, i.e. named parameters occurring after `*args` (or `*`) in the parameter list must be specified using keyword syntax in the call. This lets a function take a varying number of arguments *and* also take options in the form of keyword arguments.

In [ ]:
```
def h(a=None, *args, keyword_only=None):
    print('a: {!r}, args: {!r}, '

        'keyword_only: {!r}'
        .format(a, args, keyword_only))
```

In [ ]:
```
h.__defaults__
```

In [ ]:
```
h.__kwdefaults__
```

In [ ]:
```
h(1)
```

In [ ]:
```
h(1, 2)
```

In [ ]:
```
h(1, 2, 3)
```

In [ ]:
```
h(*range(15))
```

In [ ]:
```
h(1, 2, 3, 4, keyword_only=True)
```

In [ ]:
```
h(1, keyword_only=True)
```

In [ ]:
```
h(keyword_only=True)
```

In [ ]:
```
def h2(a=None, *, keyword_only=None):
    print('a: {!r}, '
        'keyword_only: {!r}'
        .format(a, keyword_only))
```

In [ ]:
```
h2()
```

In [ ]:
```
h2(1)
```

In [ ]:
```
h2(keyword_only=True)
```

In [ ]:
```
h2(1, 2)
```

5.2 Exercises: Functions

In [ ]:
```python
def f(*args, **kwargs):
    print(repr(args), repr(kwargs))
```

In [ ]:
```python
f(1)
```

In [ ]:
```python
f(1, 2)
```

In [ ]:
```python
f(1, a=3, b=4)
```

In [ ]:
```python
def f2(k1, k2):
    print('f2({}, {})'.format(k1, k2))
```

In [ ]:
```python
t = 1, 2
t
```

In [ ]:
```python
d = dict(k1=3, k2=4)
d
```

In [ ]:
```python
f2(*t)
```

In [ ]:
```python
f2(**d)
```

In [ ]:
```python
f2(*d)
```

In [ ]:
```python
list(d)
```

Erratum: Note that we switch from calling f2 back to calling f which may be confusing if you don't notice it.

In [ ]:
```python
f(*t, **d)
```

In [ ]:
```python
m = 'one two'.split()
```

In [ ]:
```python
f(1, 2, *m)
```

In [ ]:
```python
father = 'Dad'
```

In [ ]:
```python
locals()['father']
```

In [ ]:
```python
'Hi {father}'.format(**locals())  # A convenient hack. Only for throwaway code.
```

In [ ]:
```
def f2(a: 'x', b: 5, c: None, d:list) -> float:
    pass
```

In [ ]:
```
f2.__annotations__
```

In [ ]:
```
type(f2.__annotations__)
```

5.3 Augmented Assignment Statements

Create two names for the `str` object `123`, then from it create `1234` and reassign one of the names:

In [ ]:
```
s1 = s2 = '123'
s1 is s2, s1, s2
```

In [ ]:
```
s2 = s2 + '4'
s1 is s2, s1, s2
```

We can see this reassigns the second name so it refers to a new object. This works similarly if we start with two names for one `list` object and then reassign one of the names.

In [ ]:
```
m1 = m2 = [1, 2, 3]
m1 is m2, m1, m2
```

In [ ]:
```
m2 = m2 + [4]
m1 is m2, m1, m2
```

If for the `str` objects we instead use an *augmented assignment statement, *specifically* in-place add *+=, we get the same behaviour.

In [ ]:
```
s1 = s2 = '123'
```

In [ ]:
```
s2 += '4'
s1 is s2, s1, s2
```

However, for the `list` objects the behaviour changes.

In [ ]:
```
m1 = m2 = [1, 2, 3]
```

In [ ]:
```
m2 += [4]
m1 is m2, m1, m2
```

The **+=** in **foo += 1** is not just syntactic sugar for **foo = foo + 1. +=** and other augmented assignment statements have their own bytecodes and methods.

Let's look at the bytecode to confirm this. Notice BINARY_ADD vs. INPLACE_ADD. Note the runtime types of the objects referred to my `s` and `v` is irrelevant to the bytecode that gets produced.

In [ ]:
```
import codeop, dis
```

In [ ]:
```
dis.dis(codeop.compile_command("a = a + b"))
```

In [ ]:
```
dis.dis(codeop.compile_command("a += b"))
```

In [ ]:
```
m2 = [1, 2, 3]
```

In [ ]:
```
m2
```

Notice that `__iadd__` returns a value

In [ ]:
```
m2.__iadd__([4])
```

and it also changed the list

In [ ]:
```
m2
```

In [ ]:
```
s2.__iadd__('4')
```

So what happened when `INPLACE_ADD` ran against the `str` object?

If `INPLACE_ADD` doesn't find `__iadd__` it instead calls `__add__` and reassigns `s1`, i.e. it falls back to `__add__`.

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

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

Here's similar behaviour with tuples, but a bit more surprising:

In [ ]:
```
t1 = (7,)
t1
```

In [ ]:
```
t1[0] += 1
```

In [ ]:
```
t1[0] = t1[0] + 1
```

In [ ]:
```
t1
```

In [ ]:
```
t2 = ([7],)
t2
```

In [ ]:
```
t2[0] += [8]
```

What value do we expect t2 to have?

In [ ]:
```
t2
```

Let's simulate the steps to see why this behaviour makes sense.

In [ ]:
```
m = [7]
```

In [ ]:
```
t2 = (m,)
```

In [ ]:
```
t2
```

In [ ]:
```
temp = m.__iadd__([8])
```

In [ ]:
```
temp == m
```

In [ ]:
```
temp is m
```

In [ ]:
```
temp
```

In [ ]:
```
t2
```

In [ ]:
```
t2[0] = temp
```

For a similar explanation see https://docs.python.org/3/faq/programming.html#faq-augmented-assignment-tuple-error (https://docs.python.org/3/faq/programming.html#faq-augmented-assignment-tuple-error)

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)
```

```
In [ ]: m1
```

6 Decorators

6.1 Decorators Simplified

Conceptually a decorator changes or adds to the functionality of a function either by modifying its arguments before the function is called, or changing its return values afterwards, or both.

First let's look at a simple example of a function that returns another function.

```
In [ ]: def add(first, second):
            return first + second
```

```
In [ ]: add(2, 3)
```

```
In [ ]: def create_adder(first):
            def adder(second):
                return add(first, second)
            return adder
```

```
In [ ]: add_to_2 = create_adder(2)
```

```
In [ ]: add_to_2(3)
```

Next let's look at a function that receives a function as an argument.

Erratum: Note the trace function below has a small mistake in it so it only prints the first argument passed to the function. It should have this print instead: `print('Called {}{!r}'.format(f, args))`

```
In [ ]: def trace_function(f):
            """Add tracing before and after a function"""
            def new_f(*args):
                """The new function"""
                print(
                    'Called {}({!r})'
                    .format(f, *args))
                result = f(*args)
                print('Returning', result)
                return result
            return new_f
```

This `trace_function` wraps the functionality of whatever existing function is passed to it by returning a new function which calls the original function, but prints some trace information before and after.

```
In [ ]: traced_add = trace_function(add)
```

In [ ]:
```
traced_add(2, 3)
```

We could instead reassign the original name.

In [ ]:
```
add = trace_function(add)
```

In [ ]:
```
add(2, 3)
```

Or we can use the decorator syntax to do that for us:

In [ ]:
```
@trace_function
```

In [ ]:
```
def add(first, second):
    """Return the sum of two arguments."""
    return first + second
```

In [ ]:
```
add(2, 3)
```

In [ ]:
```
add
```

In [ ]:
```
add.__qualname__
```

In [ ]:
```
add.__doc__
```

Use `@wraps` to update the metadata of the returned function and make it more useful.

In [ ]:
```
import functools
def trace_function(f):
    """Add tracing before and after a function"""
    @functools.wraps(f)  # <-- Added
    def new_f(*args):
        """The new function"""
        print(
            'Called {}({!r})'
            .format(f, *args))
        result = f(*args)
        print('Returning', result)
        return result
    return new_f
```

In [ ]:
```python
@trace_function
def add(first, second):
    """Return the sum of two arguments."""
    return first + second
```

In [ ]:
```python
add
```

In [ ]:
```python
add.  qualname
```

In [ ]:
```python
add.  doc
```

Here's another common example of the utility of decorators. *Memoization* is "an optimization technique... storing the results of expensive function calls and returning the cached result when the same inputs occur again." -- [https://en.wikipedia.org/wiki/Memoization (https://en.wikipedia.org/wiki/Memoization)](https://en.wikipedia.org/wiki/Memoization)

Erratum: The memoize function below would be a bit clearer without the parentheses in the format: `print('Called memoized_f{!r}'.format(args))`

In [ ]:
```python
def memoize(f):
    print('Called memoize({!r})'.format(f))
    cache = {}
    @functools.wraps(f)
    def memoized_f(*args):
        print('Called memoized_f({!r})'.format(args))
        if args in cache:
            print('Cache hit!')
            return cache[args]
        if args not in cache:
            result = f(*args)
            cache[args] = result
            return result
    return memoized_f
```

In [ ]:
```python
@memoize
def add(first, second):
    """Return the sum of two arguments."""
    return first + second
```

In [ ]:
```python
add(2, 3)
```

In [ ]:
```python
add(4, 5)
```

In [ ]:
```
add(2, 3)
```

Note that this not a full treatment of decorators, only an introduction, and primarily from the perspective of how they intervene in the namespace operation of function definition. For example it leaves out entirely how to handle decorators that take more than one argument.

6.2 Exercises: Decorators

A decorator is a function that takes a function as an argument and *typically* returns a new function, but it can return anything. The following code misuses decorators to help you focus on their mechanics, which are really quite simple.

In [ ]:
```
del x
```

In [ ]:
```
def return_3(f):
    print('Called return_3({!r})'.format(f))
    return 3
```

In [ ]:
```
def x():
    pass
```

In [ ]:
```
x
```

In [ ]:
```
x = return_3(x)
```

What object will x refer to now?

In [ ]:
```
x
```

Here's equivalent code using `@decorator` syntax:

In [ ]:
```
@return_3
def x():
    pass
```

In [ ]:
```
x
```

In [ ]:
```
type(x)
```

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.

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
```

In [ ]:
```
Number.  version
```

In [ ]:
```
Number.  doc
```

In [ ]:
```
help(Number)
```

In [ ]:
```
Number.  init
```

In [ ]:
```
Number.add
```

In [ ]:
```
dir(Number)
```

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

In [ ]:
```
dir public(Number)
```

In [ ]:
```
number2 = Number(2)
```

In [ ]:
```
number2.amount
```

In [ ]:
```
number2
```

In [ ]:
```
number2.  init
```

In [ ]:
```
number2.add
```

In [ ]:
```
dir public(number2)
```

In [ ]:
```
set(dir(number2)) ^ set(dir(Number))  # symmetric difference
```

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
```

Will this work? Here's the gist of the method `add` defined above:

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

In [ ]:
```
Number.add(2)
```

In [ ]:
```
Number.add(2, 3)
```

In [ ]:
```
Number.add(number2, 3)
```

In [ ]:
```
number2.add(3)
```

Remember, here's how `__init__` was defined above:

In [ ]:
```
    def __init__(self, amount):
        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
```

In [ ]:
```
Number.__init__
```

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)
```

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)
```

In [ ]:
```
number4.mul(5)
```

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
```

In [ ]:
```
number4.add
```

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))
```

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))
```

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

In [ ]:
```python
return 5(None, None, None)
```

In [ ]:
```python
x = return 5(None, None, None)
```

In [ ]:
```python
x
```

In [ ]:
```python
type(x)
```

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

In [ ]:
```python
class y(object):  # Python 2.x
      metaclass   = return 5
```

In [ ]:
```python
class y(metaclass=return_5):  # Python 3.x
        pass
```

In [ ]:
```python
y
```

In [ ]:
```python
type(y)
```

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

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

In [ ]:
```python
return 6(None)
```

In [ ]:
```python
@return_6
class z:
    pass
```

In [ ]:
```python
z
```

In [ ]:
```
type(z)
```

7.4 Class Decorator Example

This is not a robust decorator

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

In [ ]:
```
@class_counter
class TC:   # For Python 2 use 'class TC(object):'
    pass
```

In [ ]:
```
TC.count
```

In [ ]:
```
TC()
```

In [ ]:
```
TC()
```

In [ ]:
```
TC.count
```

8 Special Methods

8.1 Special Methods of Classes

Python implements operator overloading and many other features via special methods, the "dunder" methods that start and end with double underscores. Here's a very brief summary of them, more information at [https://docs.python.org/3/reference/datamodel.html?highlight=co_nlocals#special-method-names (https://docs.python.org/3/reference/datamodel.html?highlight=co_nlocals#special-method-names)](https://docs.python.org/3/reference/datamodel.html?highlight=co_nlocals#special-method-names).

- basic class customization: `__new__`, `__init__`, `__del__`, `__repr__`, `__str__`, `__bytes__`, `__format__`
- rich comparison methods: `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`
- attribute access and descriptors: `__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`, `__get__`, `__set__`, `__delete__`
- callables: `__call__`
- container types: `__len__`, `__length_hint__`, `__getitem__`, `__missing__`, `__setitem__`, `__delitem__`, `__iter__`, (`__next__`), `__reversed__`, `__contains__`
- numeric types: `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__pow__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`
- reflected operands: `__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, `__rfloordiv__`, `__rmod__`, `__rdivmod__`, `__rpow__`, `__rlshift__`, `__rrshift__`, `__rand__`, `__rxor__`, `__ror__`
- inplace operations: `__iadd__`, `__isub__`, `__imul__`, `__trueidiv__`, `__ifloordiv__`, `__imod__`, `__ipow__`, `__ilshift__`, `__irshift__`, `__iand__`, `__ixor__`, `__xor__`
- unary arithmetic: `__neg__`, `__pos__`, `__abs__`, `__invert__`
- implementing built-in functions: `__complex__`, `__int__`, `__float__`, `__round__`, `__bool__`, `__hash__`
- context managers: `__enter__`, `__exit__`

Let's look at a simple example of changing how a class handles attribute access.

In [ ]:
```
class UppercaseAttributes:
    """
    A class that returns uppercase values on uppercase attribute access.
    """
    # Called (if it exists) if an attribute access fails:
    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))
```

In [ ]:
```
d = UppercaseAttributes()
```

In [ ]:
```
d.__dict__
```

In [ ]:
```
d.foo = 'bar'
```

In [ ]:
```
d.foo
```

In [ ]:
```
d.  dict
```

In [ ]:
```
d.FOO
```

In [ ]:
```
d.baz
```

To add behaviour to specific attributes you can also use properties.

In [ ]:
```
class PropertyEg:
    """@property example"""
    def __init__(self):
        self._x = 'Uninitialized'

    @property
    def x(self):
        """The 'x' property"""
        print('called x getter()')
        return self._x

    @x.setter
    def x(self, value):
        print('called x.setter()')
        self._x = value

    @x.deleter
    def x(self):
        print('called x.deleter')
        self.__init__()
```

In [ ]:
```
p = PropertyEg()
```

In [ ]:
```
p. x
```

In [ ]:
```
p.x
```

In [ ]:
```
p.x = 'bar'
```

In [ ]:
```
p.x
```

In [ ]:
```
del p.x
```

In [ ]:
```
p.x
```

In [ ]:
```
p.x = 'bar'
```

Usually you should just expose attributes and add properties later if you need some measure of control or change of behaviour.

8.2 Exercises: Special Methods of Classes

Try the following:

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

In [ ]:
```
g = Get()
```

In [ ]:
```
g[1]
```

In [ ]:
```
g[-1]
```

In [ ]:
```
g[0:3]
```

In [ ]:
```
g[0:10:2]
```

In [ ]:
```
g['Jan']
```

In [ ]:
```
g[g]
```

In [ ]:
```
m = list('abcdefghij')
```

In [ ]:
```
m[0]
```

In [ ]:
```
m[-1]
```

In [ ]:
```
m[::2]
```

In [ ]:
```
s = slice(3)
```

In [ ]:
```
m[s]
```

In [ ]:
```
m[slice(1, 3)]
```

In [ ]:
```
m[slice(0, 2)]
```

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

In [ ]:
```
m[::2]
```

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 [ ]:
```
generator = iter('ab')
```

In [ ]:
```
generator.__next__()
```

In [ ]:
```
generator.__next__()
```

In [ ]:
```
generator.__next__()
```

In [ ]:
```
generator.  next  ()
```

In [ ]:
```
generator = iter('ab')
```

In [ ]:
```
next(generator)
```

In [ ]:
```
next(generator)
```

In [ ]:
```
next(generator)
```

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

In [ ]:
```
generator = iter('ab')
```

In [ ]:
```
next(generator, 'z')
```

In [ ]:
```
next(generator, 'z')
```

In [ ]:
```
next(generator, 'z')
```

In [ ]:
```
next(generator, '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)
```

9.2 Exercises: Iterables, Iterators, and the Iterator Protocol

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

In [ ]:
```
it = iter(m)
```

In [ ]:
```
next(it)
```

In [ ]:
```
next(it)
```

In [ ]:
```
next(it)
```

In [ ]:
```
next(it)
```

In [ ]:
```
for n in m:
    print(n)
```

In [ ]:
```
d = {'one': 1, 'two': 2, 'three':3}
```

In [ ]:
```
it = iter(d)
```

In [ ]:
```
list(it)
```

In [ ]:
```
m1 = [2 * i for i in range(3)]
```

In [ ]:
```
m1
```

In [ ]:
```
m2 = (2 * i for i in range(3))
```

In [ ]:
```
m2
```

In [ ]:
```
list(m2)
```

In [ ]:
```
list(m2)
```

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'))))
```

9.4 Exercises: Generator Functions

Write a generator `double(val, n=3)` that takes a value and returns that value doubled n times. below are test cases to clarify.

In [ ]:
```
%load solve_double  # To display the solution in IPython
```

In [ ]:
```
from solve_double import double
def test_double():
    assert list(double('.')) == ['..', '....', '........']
    assert list(double('s.', 2)) == ['s.s.', 's.s.s.s.']
    assert list(double(1)) == [2, 4, 8]
test_double()
```

A few miscellaneous items:

In [ ]:
```
months = ['jan', 'feb', 'mar', 'apr', 'may']
```

```
In [ ]: months[0:100]
```

```
In [ ]: month num pairs = list(zip(months, range(1, 100)))
```

```
In [ ]: month num pairs
```

```
In [ ]: list(zip(*month num pairs))
```

```
In [ ]: {letter: num for letter, num in zip(months, range(1, 100))}
```

```
In [ ]: {letter.upper() for letter in 'mississipi'}
```

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()
```

10.3 Exercises: Binding Data with Functions

In [ ]:
```
import collections
```

In [ ]:
```
Month = collections.namedtuple(
    'Month', 'name number days',
    verbose=True)  # So it prints the definition
```

In [ ]:
```
Month
```

In [ ]:
```
jan = Month('January', 1, 31)
```

In [ ]:
```
jan.name, jan.days
```

In [ ]:
```
jan[0]
```

In [ ]:
```
feb = Month('February', 2, 28)
```

In [ ]:
```
mar = Month('March', 3, 31)
```

In [ ]:
```
apr = Month('April', 4, 30)
```

In [ ]:
```
months = [jan, feb, mar, apr]
```

In [ ]:
```
def month_days(month):
    return month.days
```

In [ ]:
```
month_days(feb)
```

In [ ]:
```
import operator
```

In [ ]:
```
month_days = operator.attrgetter('days')
```

In [ ]:
```
month_days(feb)
```

In [ ]:
```
month_days = operator.itemgetter(0)
```

In [ ]:
```
month_days(feb)
```

In [ ]:
```
sorted(months, key=operator.itemgetter(0))
```

In [ ]:
```
sorted(months, key=operator.attrgetter('name'))
```

In [ ]:
```
sorted(months, key=operator.attrgetter('days'))
```

In [ ]:
```
'hello'.upper()
```

In [ ]:
```
to_uppercase = operator.methodcaller('upper')
```

In [ ]:
```
to_uppercase('hello')
```