https://github.com/bcostlow/cwg_class_session_one

Idiomatic Python

Brian Costlow

About Me

Getting paid to fool around with software since 1989.

Programmed in about a dozen languages...badly...achieved mediocre in Python.

Currently independent contractor, but about to come in from the cold.

Co-organizer, Central Ohio Python User Group.

Organizer & Chair Emeritus, PyOhio Conference.

Staff Member, PyCon.

What Makes Good Code?

What Makes Good Code?

I maintain that good code has two primary properties:

1. The code does what it is supposed to do.

What Makes Good Code?

I maintain that good code has two primary properties:

1. The code does what it is supposed to do.
2. The code is written as if the maintainer who takes over your work is a psychopathic serial killer.

Idiom

noun

A group of words established by usage as having a meaning not deducible from those of the individual words (e.g., rain cats and dogs, see the light).

A form of expression natural to a language, person, or group of people.

The dialect of a people or part of a country.

Idiom

A form of expression natural to a language, person, or group of people.

```
fruits = ['apples', 'oranges', 'pears', 'apricots']


for fruit in fruits
  puts "A fruit of type: #{fruit}"
end


fruits.each do |fruit|
  puts "A fruit of type: #{fruit}"
end
```

Beginnings

```
>>>import this

>>>import antigravity

>>>from __future__ import braces
```

Two things you should grok before we get started...

1. Everything is an object.

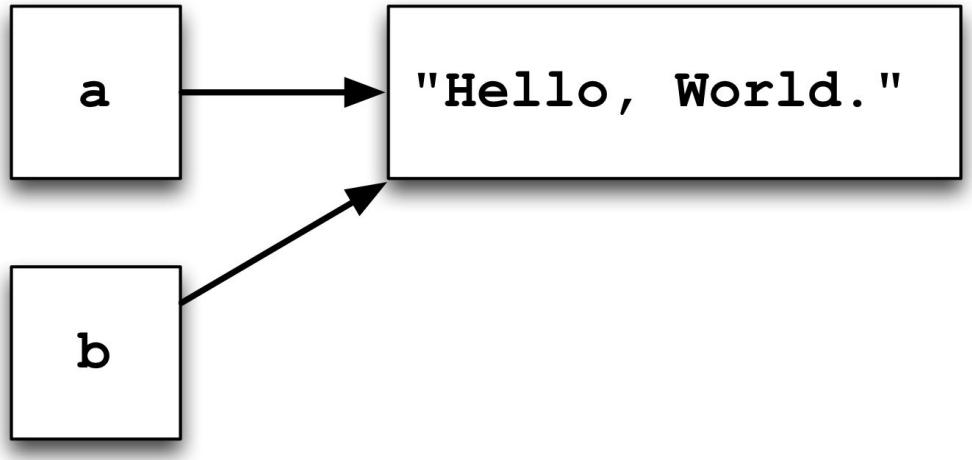
Let’s go play with the REPL a bit….

2. Names, not variables.

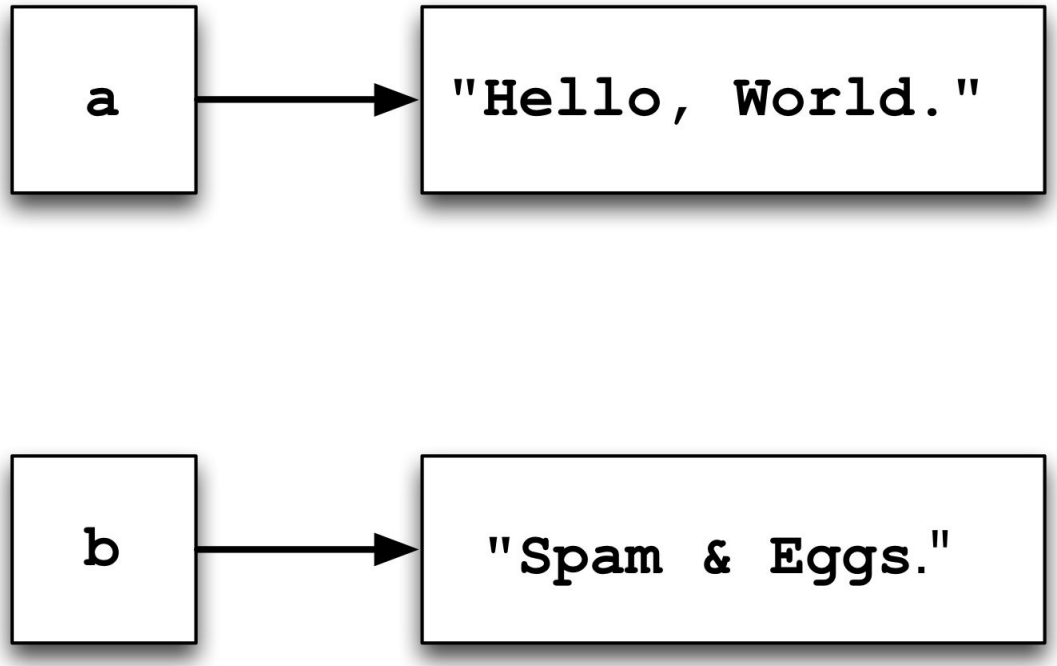
```
a = "Hello, World."
```

```
a = "Hello, World."
```


```
b = a
```



```
b = "Spam & Eggs."
```



Sort of like pointers

In that they are a reference to something.

Not exactly, in that you can't (de)reference and change the value pointed to, and both "pointers" point at the new value.

Assignment always just attaches a new label to a thing.

Note that this assignment is not the same thing as modifying a composite object!

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

b = a

b = [4, 5, 6]

print(a)

>> [1, 2, 3]

c = a

c[0] = 7

print(a)

>> [7, 2, 3]
```

Pass by reference? value?

Kind of like composite objects in Java, so if you program in that, you can think of it that way.

You are passing by value, but the value is a reference to an object.

```
foo = “spam”

bar = “eggs”

def swap(first, second):

    temp = first  # illustration only, terribly unpythonic

    first = second

    second = temp

swap(foo, bar)

print(foo)  # outputs spam

print(bar)  # outputs eggs
```

```
foo = “spam”

bar = “eggs”

def swap(foo, bar):

    temp = foo  # names don’t matter

    foo = bar

    bar = temp

swap(foo, bar)

print(foo)  # outputs spam

print(bar)  # outputs eggs
```

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

def reassign(foo):
    foo = [4, 5, 6]
reassign(foo)
print(foo)  # outputs [1, 2, 3]

def modify(foo):
    foo[0] = 7
modify(foo)
print(foo)  # outputs [7, 8, 9]
```

5. Scoping

```
foo = “spam”

def add_eggs():
    bar = foo + “ & eggs”
    print(bar)
add_eggs()  # outputs spam & eggs

def try_mod():
    foo = “eggs”
try_mod()
print(foo) # outputs spam
```

```
foo = “spam”

def do_mod():
    global foo
    foo = “eggs”
do_mod()

print(foo) # outputs eggs
```

We Now Begin Our Feature Presentation

Including the majestik møøse

Iteration

(let's look at some code)

Takeaways

Iteration is how you get things done in Python.

If you see an explicit counter while looping, there's probably a better way.

Well-written Python looks like business logic.

Generators

(more code)

Takeaways

Generators are a much more straightforward way to create your own iterators.

Generators are lazy.

Generators can be chained to provide separation of concerns.

More generators?

There's way more on generators.

***You can send values to generators* to make complex workflows.**

Generators are at the heart of coroutines in asyncio and several other green thread or async frameworks.

We don't have time to cover that stuff here. (It would take about 8 hours to cover it all).

Resources at the end.

Recursion

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Recursion is Python's kryptonite

No tail call optimization.

Everything gets pushed on the stack.

Slow.

Artificial stack limit set at 1000 in c python to prevent collisions.

Can change that limit at startup.

Code won't be portable.

High risk of segfault errors anyway.

Unpacking Sequences

You've seen this kind of code in some of the demos.

Instead of:

```
for foo in some_iterator:
```

You'll see:

```
for foo, bar in some_iterator:
```

This is an example of unpacking

```
def function_that_returns_tuple():
    return “red”, “large”
def some_other_func()
    result = function_that_returns_tuple()
    print(“You got a {}, {} tee.”.format(result[1], result[2])
def better_func()
    color, size = function_that_returns_tuple()
    print(“You got a {}, {} tee.”.format(size, color)
```

Context Managers

(code again)

More On Contexts

They are not just for IO.

They also work on locks.

You can also build them to save and restore state around an operation.

And the standard library has built-in ones for error supression, automatic closing, and standard out redirection.

```
import threading

lock = threading.lock()

try:
    lock.acquire()
    print(‘critical stuff’)
finally:
    lock.release()
```

```
import threading

lock = threading.lock()

try: # old way
    lock.acquire()
    print(‘critical stuff’)
finally:
    lock.release()
with lock: # new way
    print(‘critical stuff’)
```

Lab 1

Dictionaries

(yep, code)

Other Data Types

string and byte

In Python 2 you had ascii strings and unicode strings

```
'This is a string'

u'This is a unicode string'
```

In Python 3 you have unicode strings by default, and raw bytes.

```
'This is a unicode string'

b'this is raw bytes'
```

Lists, sets, tuples, arrays, deques

We've been using lists all along.

Lists are heterogenous sequences of any object.

Lists are mutable.

Tuples work a lot like lists, but are immutable.

Members of sets must be unique, and sets can be mutable or immutable.

Deques are list-like objects optimized for use as queues.

```
my_list = [3, 'cat', {'color': 'red', 'size': 'large'}]

my_list.append(['fred', 'barney'])

my_list[0] = 2

my_list

>> [2, 'cat', {'color': 'red', 'size': 'large'}, ['fred',
'barney']]
```

```
my_tuple = (3, 'cat', {'color': 'red', 'size': 'large'})

my_tuple[0] = 2

>> TypeError: 'tuple' object does not support item assignment

my_tuple[1]

>> 'cat'

# Note: Parens just for visual clarity, the , is the

# tuple operator

my_tuple = 3, 'cat', {'color': 'red', 'size': 'large'}
```

```
my_set = set(['red', 'green', 'red', 1, 3, 5, 4, 3, 6, 5])

my_set

>> {'red', 'green', 1, 3, 5, 4, 6}

my_set.add(5)

my_set.add('pink')

my_set

>> {'red', 'green', 1, 3, 5, 4, 6, 'pink'}

my_set[1]

>> TypeError: 'set' object does not support indexing
```

```
fr_set = frozenset(['red', 'green', 'red', 1, 3, 5, 4, 3, 6, 5])

fr_set

>> frozenset({'red', 'green', 1, 3, 5, 4, 6})

fr_set.add(5)

>> AttributeError: 'frozenset' object has no attribute 'add'
```

More on sets

Other than a quick and dirty way to insure uniqueness of members, use sets when you want to find unions and intersections between groups.

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                          # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                      # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                      # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

```
>>> a & b                    # letters in both a and b
{'a', 'c'}
>>> a ^ b                    # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Deques

Optimized for fast adding and popping from ends.

If you see stuff like this:

```
queue = list()

queue.append('foo')

queue.pop()

queue.insert(0, 'foo')

queue[0]
```

Deques

Do this:

```
from collections import deque

queue = deque()

queue.append('foo')

queue.pop()

queue.appendleft('foo')

queue.popleft()
```

Arrays

Many people (especially newbies) use the words list and array interchangeably when talking about Python.

Arrays are actually homogenous and each element is a unicode char or numeric value.

Speed optimized, but if speed of numeric manipulation is a concern, you probably want an external library.

Named Tuples

A object that behaves like a tuple, except:

The fields have names.

They can be accessed as properties of the object.

Best feature: code becomes self-documenting.

```
from collections import namedtuple

ShirtOrder = namedtuple('ShirtOrder', ['size', 'color'])


def function_that_returns_tuple():
    return ShirtOrder(“red”, “large”)
def other_func()
    order = function_that_returns_tuple()
    print(“A {}, {} tee.”.format(order.size,  order.color)
```

Comprehensions

(you guessed it, code)

Comprehension takeaways

Delimited expressions to construct certain data structures or a generator.

The core idea is for the comprehension to look more like math than code.

If you can't describe what the code does in 1 or 2 sentences, it should not be expressed as a comprehension.

Avoid too-clever one-liners.

Comprehension takeaways 2

Python has map, reduce, filter.

Prefer comprehensions unless the functional approach is clearer.

```
mat = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
# what does this do?
print([[row[i] for row in mat] for i in [0, 1, 2]])
```

Lab 2

Functions and Friends

(in the code again)

Classes & Objects

(yet another set o' code)

Lab 3

Duck Typing, Exceptions, & ABCs

(and now for something completely different)

Modules & Packages

(code)

Home Stretch

Oddities

String concatenation

```
# Never!

joined_str = ""

for string_part in very_long_list:

    joined_str += string_part

# maybe

"The " + "quick " + "fox "
```

String concatenation

```
# Most efficient for both memory use & time.

"".join(very_long_list)

"".join(["The " + "quick " + "fox"])
```

Prophylactic comma

When writing lists, dicts, tuples etc. as settings, line up vertically.

you can use a trailing comma.

```
settings = {
    "resolution": 4000,
    "orientation": "vertical",
    "color": True,
}
```

Comparisons

None, 0, and empty data structures evaluate to False.

But they are not equal to False.

Don't use comparison operators, just test the value.

Here are some examples.

```
value = ""
if value == False:
    print("This never runs")
if value == True:
    print("But Neither does this")
if value == False:
    print("This doesn't run")
else: # normally only want to run if value != ""
    print("Whoops!")
```

```
value = ""
if value:
    print("This never runs")
if not value:
    print("This does")
if value:
    print("There is a value")
else:
    print("There's not")
```

```
text_one = "cat"

text_two = "dog"

if text_one == text_two:

    # do something…

else:

    # do other thing
```

None, True & False objects

These are all singletons.

Compare them with the 'is' operator, which checks for identity.

For flow control, only need to compare None. Use True and False directly.

But comparison to Booleans useful in assert statements.

```
value = None

if value is None:

    # do stuff…

if value is not None:

    # do stuff

test_code():

    result = func_under_test()

    assert result is False
```

WAT?

```
>>> print('toys"r"us')

toys"r"us

>>> print("toys'r'us")

toys'r'us

>>> print("toys"r"us")

toysus

# WAT?
```

```
>>>42 * True
42
>>> 6 * 7 * False
0
# Originally Python used 0 & 1 for False and True.
# Initial implementation of True and False were int subtypes
# Booleans can still be used as 0 & 1 for backwards compat.
```

```
>>> 1 > 2 == False

False

# Wat?

# Python uses chained comparisons

# Above is sugar for

1 > 2 and 2 == False
```

```
>>> x = (1 << 53) + 1

>>> x + 1.0 < x

True

# WAT?
```

Not Wat

(common gotchas)

```
>>> def append_to_list(value, the_list=[]):
>>>     the_list.append(value)
>>> test_list = [1, 2, 3]
>>> append_to_list(6)
[6]
>>> append_to_list(4, test_list)
[1, 2, 3, 4]
>>> append_to_list(9)
[6, 9]
```

```
>>> def append_to_list(value, the_list=None):
>>>     if the_list is None:
>>>         the_list = []
>>>     the_list.append(value)
>>> append_to_list(6)
[6]
>>> append_to_list(9)
[9]
```

```
>>> def create_multipliers():
>>>     return [lambda x : i * x for i in range(4)]


>>> for multiplier in create_multipliers():
>>>     print(multiplier(2))
8
8
8
8
```

```
def create_multipliers():
    multipliers = []
    for i in range(4):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)
    return multipliers
```

Resources

http://www.dabeaz.com/talks.html

https://www.youtube.com/watch?v=wf-BqAjZb8M

http://rhodesmill.org/brandon/

Did we forget something?

TESTING

pip install pytest

unittest

The standard library has a unit testing module built in.

It's a j-unit style test suite.

Not really pythonic. But once it was all we had.

Python koans, a port of Ruby koans by Greg Malcolm.

https://github.com/gregmalcolm/python_koans

Great practice with Python and an intro to unittest.

pytest

But seriously,

```
pip install pytest
```

Testing is fun again, with simple functions, elegant fixtures and plain old assert statements.

http://doc.pytest.org/en/latest/

I meant the møøse

There was supposed to be a møøse!

Questions?

That's all folks...

Twitter @bcostlow

brian.costlow@gmail.com

http://cohpy.org

http://pyohio.org

https://us.pycon.org