day_zero

March 18, 2016

0.1 Pre-introduction to the class

You should start by having your class go to `url` and clone the materials for this class, either by:

1. Cloning this repo with `git clone https://`
2. Clicking the 'download zip' button on the righthand side of the page

While people are typing in urls and waiting for downloads, you can move on to:

0.2 Introduction to the class

Python is an interpreted, imperative, object oriented programming language whose primary motivation is to be easy to understand. We'll spend today talking about what each of those things mean, starting with:

0.2.1 Interpreted

Python is an interpreted language, as opposed to a compiled language. This means that, instead of being translated into a string of bits or bytes that is submitted directly to the machine, python code is submitted line by line to a program that decides what to do with each line. There are many ways to interact with this program. The simplest is:

1. Running files with python

Open up a terminal window and type this command exactly:

```
python scripts/simple.py
```

```
In [1]: ! python ../scripts/simple.py
```

```
IOKN2K!
```

Python is reading the lines in from the file simple.py, interpreting them, and then executing them. If you've taken our introduction to UNIX class, you know that to a computer, there is essentially no difference between reading commands from a file and reading them from a REPL loop.

2. You can submit commands to python via a terminal-interpreter

Python ships with a basic interpreter that you can enter by typing `python` in a terminal. This should land you in a python environment with an introductory message and a prompt that look like this:

```
Python 3.4.3 |Anaconda 2.3.0 (x86_64)| (default, Mar  6 2015, 12:07:41)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can run the file by typing `from scripts import simple`. If you look in the file, you'll see that `simple` is just running the print function. We can do this ourselves by typing:

```
In [2]: print('IOKN2K!')

IOKN2K!
```

A more popular terminal interpreter is iPython (which is developed here at Berkeley). Type `quit()` or press CNTRL+D to leave vanilla python, and once you are back in your bash terminal, type `ipython`. You should see a prompt that looks like this:

```
Python 3.4.3 |Anaconda 2.3.0 (x86_64)| (default, Mar  6 2015, 12:07:41)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

IPython is a popular option for developers who are prototyping code, and need to try out different implentations in real time. This is also true of the people of develop IPython, who report adopting a two-window setup where one window is IPython and the other is a text editor (like Vi, Sublime, or Atom). Two fantastic features of IPython are tab complete and the documentation lookup operator. Try typing `pri <tab>` into your interpreter. It should auto-complete to `print`. Add a `?` immediately after `print` and hit enter. You should see the docstring like this:

```
In [1]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

3. You can run python as a kernel in another program

The same people who make IPython also make Jupyter, which provides a notebook-like format for python similar to Mathematica or Rmd, where code, code output, text, and graphics can be combined into a single filetype that can be viewed and run by others in real time. Quit IPython (do you remember how?) and type `jupyter notebook` into your terminal. It will display some output like this:

```
[I 13:59:34.497 NotebookApp] Serving notebooks from local directory: /Users/dillonniederhut/python-for-
[I 13:59:34.497 NotebookApp] 0 active kernels
[I 13:59:34.497 NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[I 13:59:34.497 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip
```

And then will open up your default browser (or open a tab in the browser you already have running) and display the local filesystem. From here, you can start a new notebook by clicking the 'new' button on the righthand side of the page.

Important! You can't do anything in the terminal while the notebook is running.

Notebooks are not typically used for development or production, but are very common in teaching environments. For example, this teaching materials for this class were all created in Jupyter.

4. You can run python in an IDE

IDE stands for 'Integrated Development Environment', and is a graphical user interface that typically includes an output window, a text editor with built-in `run` and `debug` functions, display windows for plots and filesystems, and some amount of declaration tracking. In this class, we'll be using Rodeo, a lean IDE that uses IPython as its interpreter. There are many other choices, including: * IDLE - Python's built-in IDE (no one really uses this) * Spyder - IDE that ships with Anaconda (similar to Rodeo) * PyCharm - JetBrain's IDE (feature-heavy; includes VCS support and cross-referencing) You should already have Rodeo installed - double click the icon (wherever you put it) to start up the program.

0.3 Object Orientation

Earlier, we said that python is an object oriented language. This means that python things about it's code the same way that you think about the stuff around you. In the grand scheme of computer software, object orientation is a way of organizing code such that it is easy to update without breaking. This means grouping functions that serve a similar purpose into hierarchies. However, stating it this way is confusing and abstract.

You can think about it this way: a soccer ball is an object. So is a basketball. They share a lot of things in common. It's simpler to know that balls generaly bounce than to explicitly declare for every ball I ever see in my entire life whether it bounces or not. I can't bounce you, for example, but you didn't need to tell me that when I met you. If I came to believe that people were bounce-able, I would update my idea of people generally, not every person specifically.

```
In [3]: type(4)
```

```
Out[3]: int
```

We call things like you and basketball objects, and they are in classes like human and ball. If I want to create a new object, like a football, I don't have to declare every single thing there is to know about footballs. I can say it inherits attributes from the class ball, except that it's an oblate spheroid instead of a sphere. Easy.

In python, things like numbers are a class of objects. A specific number, however, needs a name. In much the same way, if I want to talk to you about the deflated Patriots' football, I can't just ask you about 'the ball' and expect you to know what I mean. In python, we call the specific ball under question an instance, and it needs a unique name for the duration of our discussion.

```
In [4]: four = 4
```

```
In [5]: type(four)
```

```
Out[5]: int
```

```
In [6]: four + 4
```

```
Out[6]: 8
```

If I assign something else the name `four`, it overwrites the instance that `four` previously referred to.

```
In [7]: four = 5
        four + 4
```

```
Out[7]: 9
```

Because everything in python needs to have a unique name, managing what names are defined at any time becomes very important. Python comes with built in names like `print` and `open` that are already taken. Other functions and libraries don't exist in Python on their own, but need to be brought in with a function called `import`. As a simple example, let's import a library for math called `math`.

```
In [8]: import math
```

Now type in `math.` and press tab.

```
math.acos        math.acosh       math.asin        math.asinh
math.atan        math.atan2       math.atanh       math.ceil
math.copysign    math.cos         math.cosh        math.degrees
math.e           math.erf         math.erfc        math.exp
math.expm1       math.fabs        math.factorial   math.floor
math.fmod        math.frexp       math.fsum        math.gamma
math.hypot       math.isfinite    math.isinf       math.isnan
math.ldexp       math.lgamma      math.log         math.log10
math.log1p       math.log2        math.modf        math.pi
math.pow         math.radians     math.sin         math.sinh
math.sqrt        math.tan         math.tanh        math.trunc
```

What happened? What is the purpose of hiding `log10` hidden behind `math`?

All the names in use at any time are called your 'namespace'. Keeping functions in dot notation behind their library keeps you from polluting your namespace, or accidentally overriding other variables.

> side note - if you are coming from R, the dot naming convention, e.g. `my.data`, can never be used because of this

Dot notation doesn't only apply to objects in a library, it is also used for functions that are attached to an object (these are called 'methods'). Try tab complete on `four`.

```
four.bit_length    four.conjugate     four.denominator   four.from_bytes
four.imag          four.numerator     four.real          four.to_bytes
```

You won't see `+` or `-` in the methods (they are actually there, just hidden from the user) because `four.add(5).add(6)` is less easy to read and understand than `four + 5 + 6`. Easy-to-understand code is the main design principle behind the python language. In fact, you can import the python philosophy into your session the same way you would import anything else.

```
In [9]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Guido's insight in creating python was that code is read more frequently than it is written, so writing code that is easy to read should be a major principle in the design of the language. All that stuff about favoring explicit actions is so that someone reading your code isn't missing important stuff that is happening, but not written into the code in an obvious way.

> side note - if you are a cool cat, you abbreviate Guido's name as `GvR`

> side note - if you are a really cool cat, you call yourself a pythonista

> side note - GvR named the python language after Monty Python, which should tell you something about pythonistas

Likewise, the line about having only one way to perform an action makes code much easier to read. For example, you'll learn tomorrow about how to read and write to disk (so don't worry about taking notes on this). There are many ways that you *could* do this, but in python the *correct* way is:

```
with open(filepath, 'r') as f:
    my_data = f.read()
```

Any time you see code that looks something like this, you know exactly what it is doing, even if you haven't seen it before. For example, what do you think this code does?

```
with open(filepath, 'r') as f:
    my_data = json.load(f)
```

While we've assigned objects to names, we haven't really made them do much yet. Any object that modifies data, whether this is returned to the user or happens behind the scenes, is called a function. In python, functions are designated by parens attached to the object name.

```
In [10]: math.sqrt(four)

Out[10]: 2.23606797749979
```

If you call a function without parens, python will print something about the function.

```
In [11]: math.sqrt

Out[11]: <function math.sqrt>
```

0.4 Data types

Programming is all about data, and any given programming language will have different ways of dealing with different kinds of data. The constraints on how a programming language deals with data come from both the hardware and the users. On the hardware side, a computer operates on data at the binary level, so everything needs to be fundamentally composed of `1`s and `0`s. On the user side, manipulating numbers is (and should be!) very different from manipulating words. Python has five basic types of data.

> side note - unlike many other languages, you do not need to tell python what type your data is (although you can anyway, and it is often a good idea to be 'defensive' about typing).

0.4.1 1. Integers

We've already seen some of these. An integer in Python is exactly what it sounds like:

```
In [12]: type(1)
```

```
Out[12]: int
```

You can perform all of the basic operations on integers that you expect, like basic arithmetic:

```
In [13]: 3 + 2
```

```
Out[13]: 5
```

```
In [14]: 3 - 2
```

```
Out[14]: 1
```

```
In [15]: 3 * 2
```

```
Out[15]: 6
```

```
In [16]: 3 / 2
```

```
Out[16]: 1.5
```

This last result should be very surprising to you if you come from a language like C++ or Java (or even an older version of Python!) - we just divided two integers and got something else! As of python 3, float division is standard even when the datatypes are integers. If you want integer division or integer modulus, you need to use `//` and `%`:

```
In [17]: 3 // 2
```

```
Out[17]: 1
```

```
In [18]: 3 % 2
```

```
Out[18]: 1
```

You can also perform logical comparisons on integers, which return another kind of value (note that equality testing is done with two equal signs):

```
In [19]: 3 > 2
```

```
Out[19]: True
```

```
In [20]: 3 == 2
```

```
Out[20]: False
```

```
In [21]: 3 != 2
```

```
Out[21]: True
```

Integers are often used in programming to count the number of times something has happened. In this case, you would initialize a variable with a value of zero:

```
In [22]: counter = 0
```

and then increment it:

```
In [23]: counter += 1
         print(counter)
```

```
1
```

Run the code again. What happened? How do you think you would decrement a value?

0.4.2 2. Booleans

Since everything in python is an object, and every object must belong to a class, the `True`s and `False`s that we are getting also have a type and associated methods.

```
In [24]: type(True)
```

```
Out[24]: bool
```

Principally, bools are used for decision making, which you'll learn about tomorrow. They are also often used to indicate whether an attempt at doing something was successful or not. Bools can be evaluated in logic tables:

```
In [25]: not True
```

```
Out[25]: False
```

```
In [26]: True and False #or True & False
```

```
Out[26]: False
```

```
In [27]: True or False #or True | False
```

```
Out[27]: True
```

Internally, python stores values for `bool` type objects as a binary value, which means you can do some weird things with `True` and `False`

```
In [28]: True * 3
```

```
Out[28]: 3
```

```
In [29]: 4 / False
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-29-f1cbcd91af4b> in <module>()
----> 1 4 / False


ZeroDivisionError: division by zero
```

Sometimes this works the way you want:

```
In [30]: 1 and True
```

```
Out[30]: True
```

But sometime it does not:

```
In [31]: True and 1
```

```
Out[31]: 1
```

0.4.3 3. Floats and type coersion

Floating point numbers are python's way of handling numbers that can't efficiently be represented as integers, either because they are very large or because they are inherently rational. In python, you indicate that you want a number to be a float with a `.`

```
In [32]: type(1.)
```

```
Out[32]: float
```

Most of the numerical data you'll process will be as floating point numbers, which behave pretty much the same as integers in mathematical operations, but come with a few extra methods.

```
In [33]: 3.5 + 2.5
```

```
Out[33]: 6.0
```

```
In [34]: 3.5 + 2.5
```

```
Out[34]: 6.0
```

```
In [35]: math.pi.as_integer_ratio()
```

```
Out[35]: (884279719003555, 281474976710656)
```

The ability to efficiently represent complex numbers comes with a risk of imprecision, which grows for larger numbers.

```
In [36]: 100.2 - 100
```

```
Out[36]: 0.20000000000000284
```

```
In [37]: 1000000000.2 - 1000000000
```

```
Out[37]: 0.20000004768371582
```

This can land you in trouble when making comparisons:

```
In [38]: 100.2 - 100 == 0.2
```

```
Out[38]: False
```

When you mix integers and floating point number in a calculation, python casts the result as a float, even if the result is an integer

```
In [39]: type(0.5 * 2)
```

```
Out[39]: float
```

```
In [40]: type(3/2)
```

```
Out[40]: float
```

You can coerce floating point numbers into integers, but note that you lost information when you do this.

```
In [41]: int(4.5)
```

```
Out[41]: 4
```

What you might not have guessed is that you can also convert floating point numbers into True and False. Like JavaScript, Python has 'truthiness', which means that non-Boolean values can evaluate to `True` and `False` in certain situations. This is done to avoid obtuse syntax, like:

```
if number_of_students != 0:
    have class
```

You'll see this more tomorrow, but just to introduce it now:

```
In [42]: number_of_students = 0.
         if number_of_students:
             print('Class is in session!')
```

Floating truthiness is that 0 is always `False`, but everything else (including negative numbers) is `True`.

```
In [43]: number_of_students = -1.
         if number_of_students:
             print('Class is in session!')

Class is in session!
```

Now let's try a small challenge To check that you've understood this conversation about data types, objects, and ways to interact with python, we're going to have you do a small test challenge. Partner up with the person next to you - we're going to do this as a pair coding exercise - and choose which computer you are going to use.

In a text editor or IDE on that computer, open `challenges/00_introduction/A_objects.py`. This is a python script file that you can run from the command line.

In the file are comments describing some tasks. When you think you've completed them successfully, open a terminal window and navigate to `challenges/00_introduction`, then type `py.test test_A.py` and hit enter.

> students may need to install pytest with `conda install pytest` or `pip install pytest`

If you have completed everything successfully you will see:

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /Users/dillon/Dropbox/dlab/workshops/pyintensive/challenges/00_introduction, inifile:
collected 2 items

test_A.py ..

=========================== 2 passed in 0.01 seconds ===========================
```

If you have not, you'll see something like this:

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /Users/dillon/Dropbox/dlab/workshops/pyintensive/challenges/00_introduction, inifile:
collected 2 items

test_A.py .F

=================================== FAILURES ===================================
_____________________________________ test_dillon _____________________________________
```

```
    def test_dillon():
>       assert isinstance(float, A.dillon)
E       AttributeError: module 'A_objects' has no attribute 'dillon'

test_A.py:12: AttributeError
======================== 1 failed, 1 passed in 0.01 seconds ========================
```

with information about which test failed and why. In this case, testing the object `dillon` failed because A_objects.py does not contain an object with the name `dillon`.

0.4.4 4. Strings and containers, part -1

Srings are how Python represents character data like "Dillon" or "It was the best of times, it was the worst of times".

```
In [44]: "A string can be in double quotes"

Out[44]: 'A string can be in double quotes'

In [45]: 'Or single quotes'

Out[45]: 'Or single quotes'

In [46]: 'As long as ya'll are careful with "apostrophes" and quotations'


          File "<ipython-input-46-63726c22d368>", line 1
        'As long as ya'll are careful with "apostrophes" and quotations'
                        ^
    SyntaxError: invalid syntax
```

Just like with integers and floats, you can specify types with a function call. Just about anything can be coerced to a string:

```
In [47]: str(4.0)

Out[47]: '4.0'

In [48]: str(True)

Out[48]: 'True'
```

Internally, these are represented as bytes (which you can also access, but probably don't want to). Translating from bytes to string literals is known as "decoding", and translation in the other direction is called "encoding".

Why am I telling you this? Because if you are here for web scraping or any kind of text analysis, you will immediately run into encode/decode errors. The issue here is that there are approximately one bajillion ways to convert between machine readible bytes and human readible characters. This means that some characters don't exist in some encodings:

```
In [49]: ''.encode('ascii')


        ---------------------------------------------------------------------------

        UnicodeEncodeError                        Traceback (most recent call last)
```

```
    <ipython-input-49-7d97065a2a3c> in <module>()
----> 1 ''.encode('ascii')


    UnicodeEncodeError: 'ascii' codec can't encode character '\xe9' in position 0: ordinal not in ra
```

It also means that the the same character has a one-to-many mapping with bytes:

```
In [50]: ''.encode('utf-8')

Out[50]: b'\xc3\xa9'

In [51]: ''.encode('iso-8859-1')

Out[51]: b'\xe9'
```

The encoding for any kind of string data depends on a combination of:

1. The program that created it
2. The operating system that hosts the program
3. The filetype where it was written to disk

Infuriatingly, the encoding of characters is not always declared in a file, especially if the file was written some time before 2005.

As a general rule, the characters on English keyboard keys are the same in all encodings. Most things `UNIX` and Python are either `ascii` or `utf-8`, which is forwards-compatible with ascii. If the file doesn't declare its encoding anywhere or it is really old, it is probably `iso-8859-1`, which is the American/Western-Europe encoding in Microsoft.

Python has rich methods for string manipulation, even in the standard library, which makes it a popular language for text analysis. To get started, what do you think will happen if we use the `+` operator on two strings?

```
In [52]: 'Juan' + 'Shishido'

Out[52]: 'JuanShishido'
```

Or the '*' operator?

```
In [53]: 'Juan' * 3

Out[53]: 'JuanJuanJuan'
```

The `-` operator won't work. Pretend you are `GvR` and tell me why.

```
In [54]: 'Andrew' - 'Chong'


    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-54-7e6e2e406762> in <module>()
----> 1 'Andrew' - 'Chong'


    TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

There isn't a clear meaning behind subtracting a string from another string. Do we want one 'Chong' removed from Andrew? All of them? Or all of the individual characters in 'Chong'? This would need to be implicit in the code somewhere - no good!

If you want to remove part of a string, you'll need to use a substitution method like:

```
In [55]: my_string = 'Dav Clark wears a beret'

         my_string.replace('beret', '')

Out[55]: 'Dav Clark wears a '
```

Of course, you could replace beret with something else

```
In [56]: my_string = my_string.replace('beret', 'speedo')
         print(my_string)

Dav Clark wears a speedo
```

Just like floats, strings are also truthy. In this case, a true string is just one that isn't empty:

```
In [57]: bool(my_string)

Out[57]: True

In [58]: bool('')

Out[58]: False
```

Simple string transformations are easy in Python

```
In [59]: my_string.lower()

Out[59]: 'dav clark wears a speedo'

In [60]: my_string.title()

Out[60]: 'Dav Clark Wears A Speedo'
```

Each transformation has an associated test

```
In [61]: my_string.isupper()

Out[61]: False
```

You can count the number of substrings in a string

```
In [62]: my_string.count('e')

Out[62]: 3
```

Which means you can say

```
In [63]: bool(my_string.count('e'))

Out[63]: True
```

But that's weird to read, so instead we would want to write:

```
In [64]: 'e' in my_string
```

```
Out[64]: True
```

This works because strings in Python are technically containers for characters (an empty string – '' – is just a container with no characters in it. Because strings are containers, this means that each character has an index value. You can get the index value of a substring with:

```
In [65]: my_string.find('speedo')

Out[65]: 18
```

The `18` here is giving you the index of 'Dav'. If we look for a string that isn't there, we see something a little unexpected.

```
In [66]: my_string.find('Dillon')

Out[66]: -1
```

This tells us two things:

1. Dillon does not wear speedos
2. We can't use str.find() as a truthy value

To find out why (why 2; why 1 should be self-explanatory), let's see how to grab things by index.

```
In [67]: my_string[18]

Out[67]: 's'
```

This gives us the `s` in `speedo`. You can grab more than one character by specifying a beginning and an end to the index like this:

```
[start:end]
```

Let's imagine we wanted to grab the whole word. How would we do that?

```
In [68]: my_string[18:18+len('speedo')] # or my_string[18:18+6]

Out[68]: 'speedo'
```

You might have tried the following, which does not work:

```
In [69]: my_string[18:18+len('peedo')] # or my_string[18:18+5]

Out[69]: 'speed'
```

The reason is that python indices are only inclusive on one end. Mathematically, this is written as `[x,y)`. This keeps you from getting overlapping parts of a string when subsetting more than once, and makes it really easy to grab substrings just with

```
[i : i + len(s)]
```

because the distance between two points of an index is the same as the length of the object.
Grab 'Dav' from `my_string`.

```
In [70]: my_string[:3]

Out[70]: 'Dav'
```

The index starts at zero! Python is a 'zero-indexed' language, like most computer languages (but unlike R). This lets us grab items out of the start of a container just by knowing how long they are. Unfortunately, it means that if we call `str.find()` on 'Dav', it returns a position of `0`, so we can't coerce these results into a bool.

For text analysis, you typically don't analyze entire containers of characters. More likely, you'll want to split strings on one of two features:

```
In [71]: "It was the best of times \nIt was the worst of times".split('\n')

Out[71]: ['It was the best of times ', 'It was the worst of times']
```

If you don't specify what character to split on, Python uses whitespace by default.

```
In [72]: my_string.split()

Out[72]: ['Dav', 'Clark', 'wears', 'a', 'speedo']
```

These both turn strings (which remember, are containers), into a container of containers called a `list`. You'll learn more about these tomorrow.

0.4.5 5. Functions, the not-exactly-data-datatype

As an object oriented language, functions in Python are also objects that can be assigned to names and passed to other functions.

```
In [73]: type(max)

Out[73]: builtin_function_or_method
```

Unlike other datatypes, functions in python need to be created with a keyword – `def`. This is a normally thing in OOLs, but seems odd in Python because it lacks the `val` and `var` keywords for creating data.

```
In [74]: def increment(x):
             return x + 1

         increment

Out[74]: <function __main__.increment>

In [75]: increment(4)

Out[75]: 5
```

When you run a function, it creates its own namespace to keep any object names in the function from insulated from object names in the global environment. Imagine if every time you wanted to have a conversation, you had to invent new words for everything you wanted to talk about – super dangerous! Namespaces help to enforce modularity in software, to keep functions from breaking when other things change.

To see how this works, let's modify that increment function a little bit

```
In [76]: def increment(x):
             n = 1
             return x + n
         increment(1)

Out[76]: 2

In [77]: n = 9000
         increment(1)
```

```
Out[77]: 2
```

You can also use a function to create other functions.

```
In [78]: def make_incrementor(n):
             def incrementor(x):
                 return x + n
             return incrementor

         chapman = make_incrementor(-2)
         chapman

Out[78]: <function __main__.make_incrementor.<locals>.incrementor>

In [79]: chapman(5)

Out[79]: 3
```

You can also also give functions to other functions, just like any other kind of data. We have done this already by calling `type` on a function, but we can to this ourselves as well.

```
In [80]: def my_apply(x, fun):
             return fun(x)
         my_apply

Out[80]: <function __main__.my_apply>

In [81]: my_apply(-1, chapman)

Out[81]: -3
```

Now let's try another small challenge! Pair up with your partner again - but this time, use the other person's computer. You are going to try the next challenge for today, which is in `challenges/00_introduction/B_syntax.py`.

When you think you have met the challenge, run `py.test test_B.py`. If you don't pass the tests, be sure to pay attention to the error messages!