

A Rapid and Informal Introduction to Python for Data Science

November 4th, 2016, Cal State Fullerton
Center for Computational and Applied Mathematics

Brian Vegetabile, PhD Candidate
Department of Statistics
University of California, Irvine



Workshop Outline

- 2:00PM - 3:45 *Overview & Basics of the Python Language*
- 3:45PM - 4:00 *Break*
- 4:00PM - 4:50 *Python and Data Science - Overview of some simple libraries*
- 4:50PM - 5:00 *Break*
- 5:00PM - ? *Simple Data Analysis Example*

What this workshop is...

- Since this is a four hour workshop it is tough to decide what gets taught and what doesn't
- In that pursuit this course is intended as an introduction to the Python programming language
 - Focus on the fundamentals of the Python language with the goal of using Python for working with data
- This course is not intended to teach “how to program”
 - Assume a basic familiarity with “programatic” thinking
 - This does not mean that if you don't know how to program the workshop will be useless, but it definitely will not make someone a programmer in four hours
- The ideal student then has some experience with programming or “programatic thinking” and may have written codes in other languages (Matlab, R, C, C++, Java) to analyze data.

What is Python?

- Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java.
- Some notable features
 - Elegant easy to read syntax
 - Easy to use programming language, ideal for prototyping or ad-hoc development projects
 - Rich standard library for performing a variety of tasks
 - Runs anywhere: Mac, Windows, Linux, Unix
 - Free to use: Free to download and use in development projects, copyright available under an open source license

Downloading Python:

Understanding Different Distributions of Python

- Python can be downloaded from the official python.org
- Additionally, there are many “alternative” distributions
 - IronPython - Python running on .NET
 - ActiveState ActivePython
 - **Enthought Canopy - Commercial distribution for scientific computing**
 - **Anaconda Python - a full Python distribution for data management, analysis and visualization of large data sets**

Anaconda Python

- Many people often install the Enthought Canopy or the Anaconda Python distributions
 - We'll be using the Anaconda Python distribution today
- What is Anaconda?
 - Free, easy-to-install **package manager**, environment manager, Python distribution
 - Collection of over 720 open source packages with free community support
 - Platform-agnostic, can be used on Windows, OS X and Linux

Installing Anaconda

- If you haven't already, go to the following website and follow the instructions to install Anaconda
 - <https://www.continuum.io/downloads>
- **Ensure that you have installed Anaconda for Python 2.7**
 - This process will take a little bit, try to follow along while this is installing

References for Continuing Learning

- **List of Python Beginners Guides (Non-Programmers)** - <https://wiki.python.org/moin/BeginnersGuide/NonProgrammers>
- **List of Python Beginners Guides (Programmers)** - <https://wiki.python.org/moin/BeginnersGuide/Programmers>
- **Codecademy** - <https://www.codecademy.com/learn/python>
- **Udacity** - <https://www.udacity.com/course/programming-foundations-with-python--ud036>

References for this Workshop

- **Programming in Python 3: A Complete Introduction to the Python Language**
 - <https://www.qtrac.eu/py3book.html>
 - While the book is focused on Python 3 it has been a great reference for me when I began diving deeper into the language
 - Many of the examples here are based off of similar examples from this book
- **Python documentation:** <https://docs.python.org/2/>

Python Overview - Getting Started

Interfacing with Python - Command Line

- There are a few ways to interface with Python
- The first and easiest way is directly from the command line in an interactive fashion
 - “Glorified calculator mode”
- Let’s get started...
- Open a new terminal and type `i p y t h o n`, hit enter and then type:
 - ```
In[1]: myname = 'Brian'
```

```
In[2]: print 'Hello, my name is {0}'.format(myname)
```

# Interfacing with Python - Command Line

---

- Should have returned something like:
  - `Hello, my name is Brian`
- This may look odd to those of you unfamiliar with Python
- The first line created a variable called `myname` and then provided it with the value `'Brian'`
- The second line is composed of two parts
  - A `print` statement, and...
  - A string `'Hello, my name is {0}'` modified by the method `.format()` which has been passed the argument `myname`

# Everything's an object

---

- We're getting a little ahead of ourselves here, but let's start explaining a bit about what is going on
- What is odd about this is the way that we were able to modify the string to include the variable `myname`

# Everything's an object

---

- In Python everything is an object
  - This means that it carries with it special attributes and methods once it has been specified
  - Throughout today we'll introduce this type of thinking within Python which allows you to flexibly interact with code.
  - Once it becomes familiar to you it will be second nature.
- The syntax for this is
  - `object.attribute` # Variables of the object
  - `object.method()` # Functions available to the object

# Interfacing with Python - Scripts

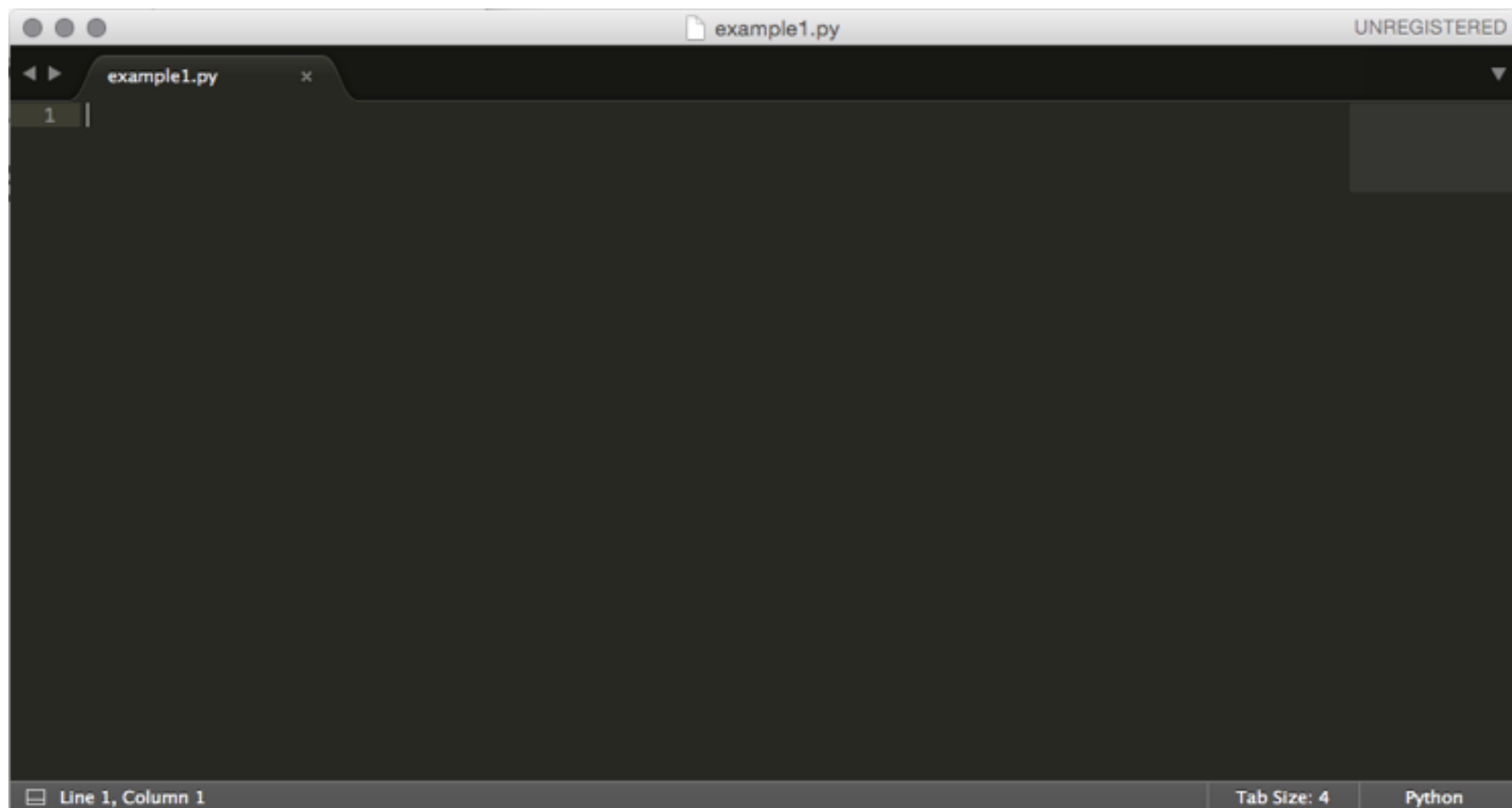
---

- A more advanced method is to begin to write scripts within text files. Let's create a file first...
- From the command line on Mac/Linux, type:
  - ```
cd  
cd Desktop/  
mkdir 1104_pyworkshop  
cd 1104_pyworkshop/  
touch example1.py
```
- From the command prompt on Windows, type:
 - ```
cd %HOMEPATH%
cd Desktop/
mkdir 1104_pyworkshop
cd 1104_pyworkshop/
type NUL >> example1.py
```

# Interfacing with Python - Scripts

---

- From your Desktop open the folder. The file that was created should be there. Now open the file with a text editor.

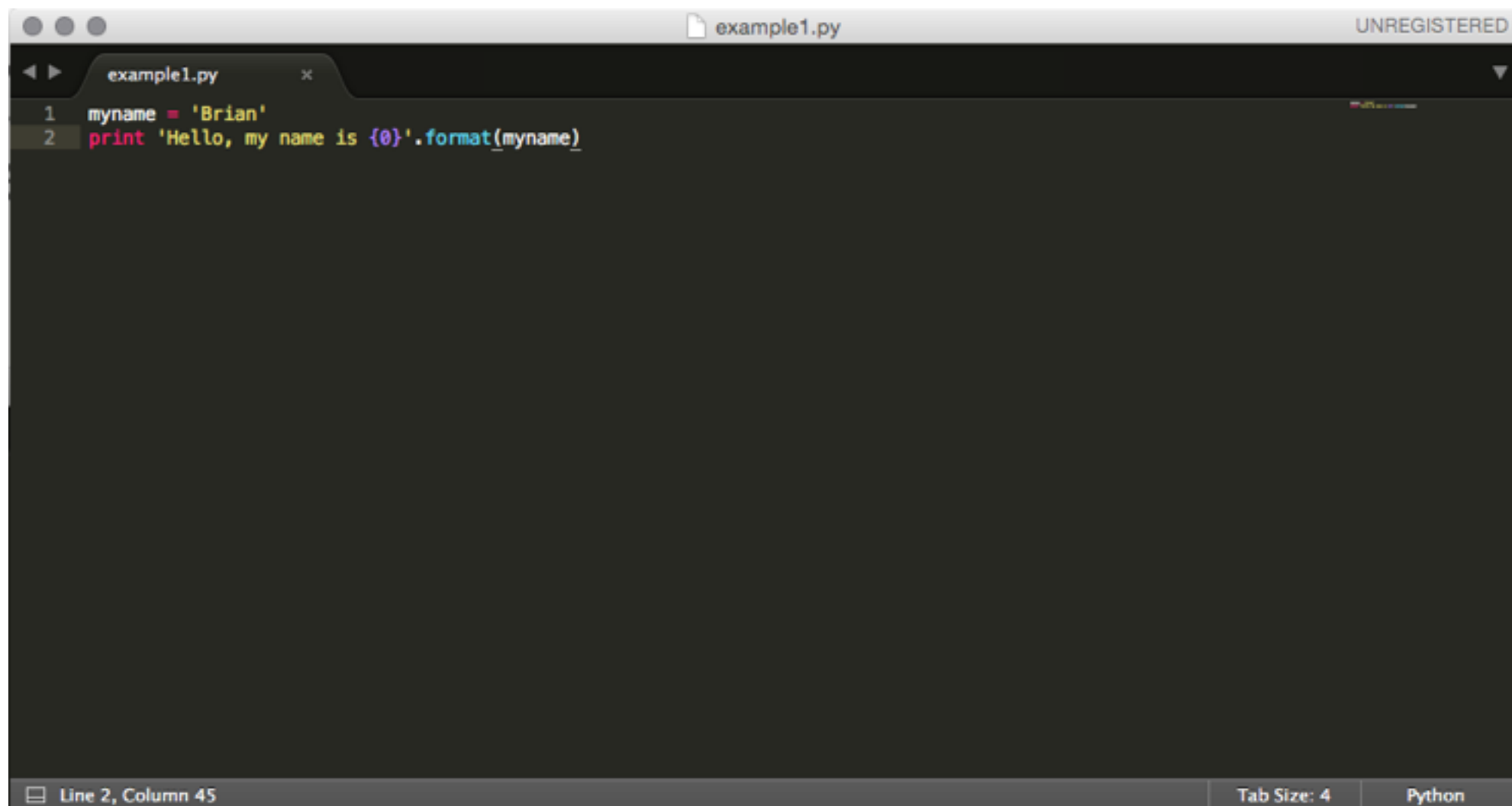




# Interfacing with Python - Scripts

---

- Add the code that we used in the interactive mode to the script. It should look as follows:
  - Colors may vary...



The screenshot shows a code editor window titled 'example1.py' with a status bar indicating 'UNREGISTERED'. The editor contains two lines of Python code:

```
1 myname = 'Brian'
2 print 'Hello, my name is {}'.format(myname)
```

The status bar at the bottom shows 'Line 2, Column 45', 'Tab Size: 4', and 'Python'.

# Interfacing with Python - Scripts

---

- Now let's run the script...
- First check that the file is within the current directory. From command line on a Mac/Linux
  - `pwd; ls`
- Or on a Windows machine type:
  - `cd & dir`
- The simple way to run a program that takes no arguments is by typing `ipython filename.py`  
Therefore if the file is within this directory type:
  - `ipython example1.py`
- If you were successful you should see the following


Hello, my name is Brian

# How do Python programs work?

---

- The way a Python program is structured provides instructions to the computer as to the order in which to execute code
- Each statement encountered in a .py program is executed in the order in which they are arrived at
- The order of commands can be diverted or “controlled” using certain operations and logical operations
- Additionally the structure of the spacing provides information to the compiler as to which parts of the code are organized together

# How do Python programs work?



```
counter_example.py ×
1 # Example of a simple counter function
2
3 counter = 0
4
5 print 'Starting Counter Position: ' + str(counter)
6
7 while counter <= 1000:
8 counter += 1
9 if not counter % 100:
10 print 'Counter : '.rjust(15) + str(counter)
11
12
13 print 'Final Counter Position : ' + str(counter)
```



# Interfacing with Python - Jupyter Notebooks

---

- Scripts are a simple and straight forward way to create python executable code, but may not be the easiest way to test small bits of code
- Additionally, they are not as useful if you're mainly trying to perform data analysis tasks where you want to consistently view the data in different ways
- An alternative way to interface with python is with Jupyter notebooks (previously referred to as iPython Notebooks)



# Interfacing with Python - Jupyter Notebooks

---

- <http://jupyter.org/>
  - Open source, interactive data science and scientific computing across over 40 programming languages.
  - The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text.
  - Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.
- Installing Jupyter Notebooks is easy! (It was already installed with Anaconda!)



# Interfacing with Python - Jupyter Notebooks

- Getting a Jupyter notebook started

- From command line type

```
jupyter notebook
```

- Two things will happen:

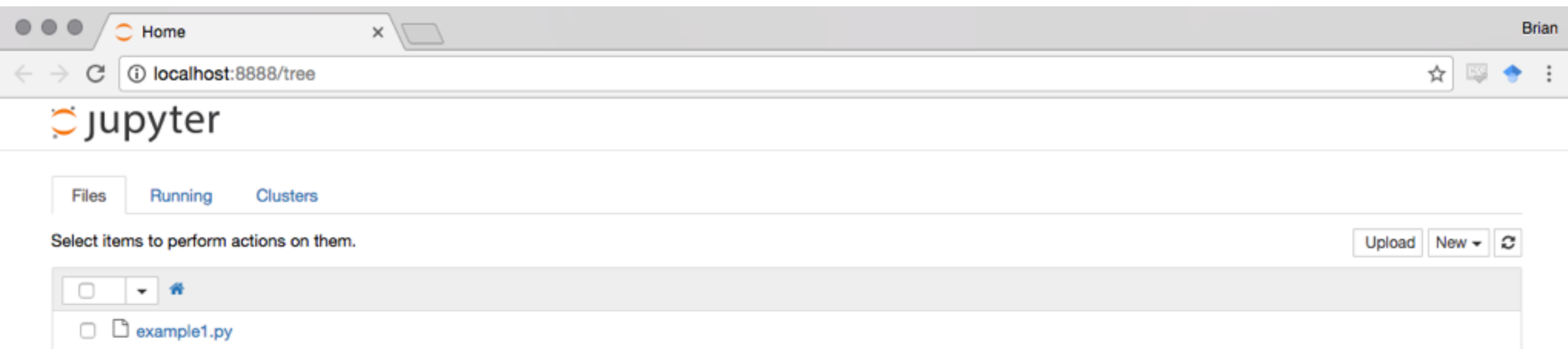
1. First terminal will begin executing some 'commands'. Basically a local server to run the Jupyter notebook

```
1104_pyworkshop — python2.7 — 122x24
bvegetabile ~/Desktop/1104_pyworkshop$ jupyter notebook
[I 11:31:25.256 NotebookApp] Serving notebooks from local directory: /Users/bvegetabile/Desktop/1104_pyworkshop
[I 11:31:25.257 NotebookApp] 0 active kernels
[I 11:31:25.257 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 11:31:25.257 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
^
```



# Interfacing with Python - Jupyter Notebooks

- Two things will happen:
  1. First terminal will begin executing some 'commands'
  2. A web browser will open showing a directory structure
    - Allows access to files in the directory where the command was execute and all subdirectories of the original directory

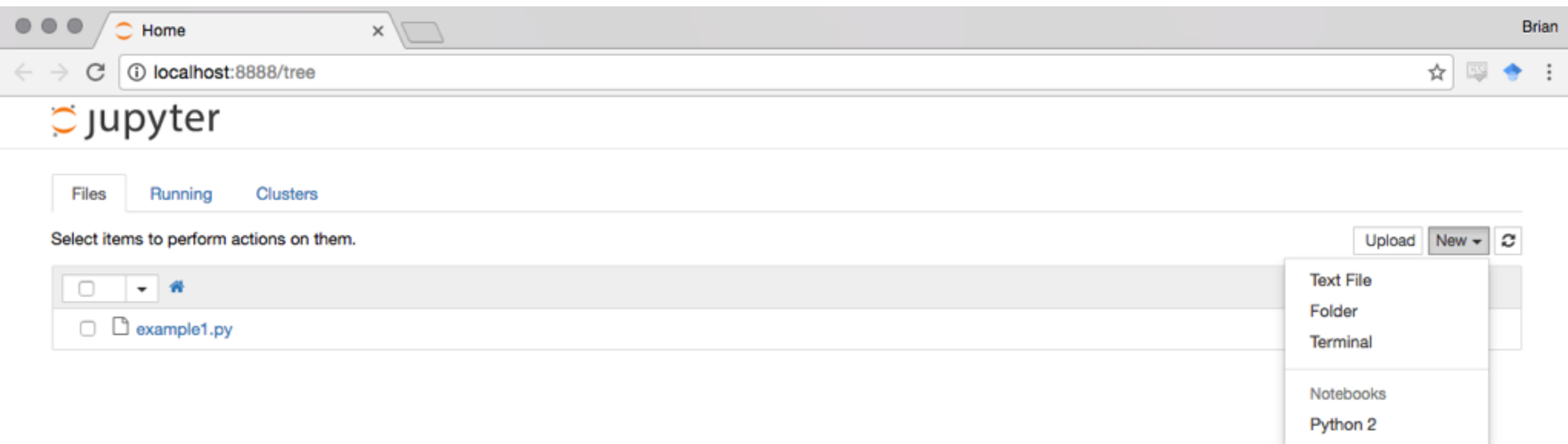






# Interfacing with Python - Jupyter Notebooks

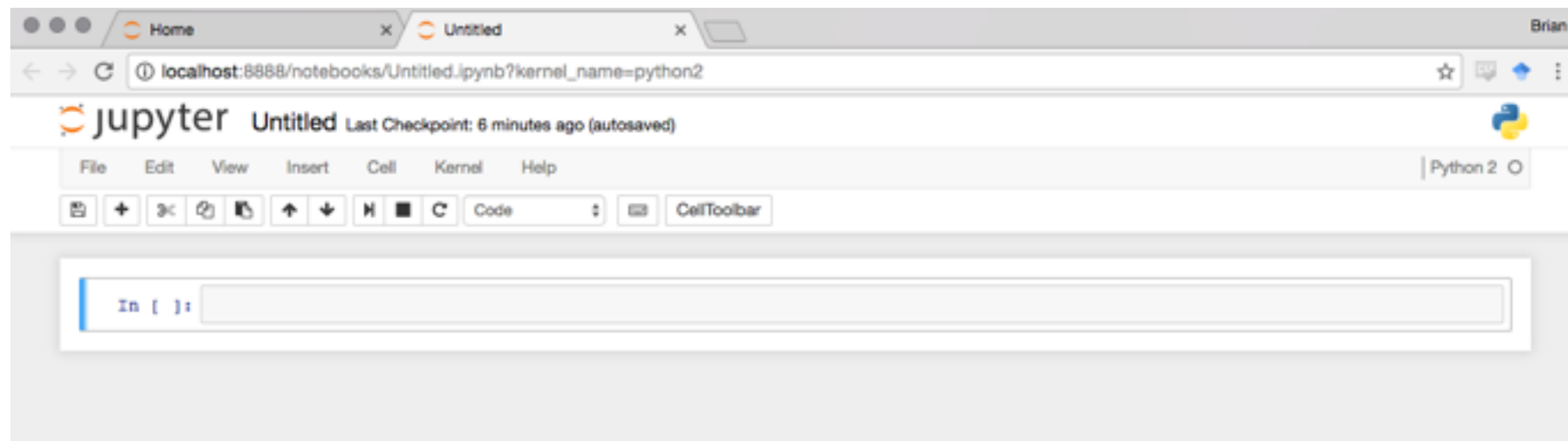
- We'll now create a new notebook
  - Click New > Python 2



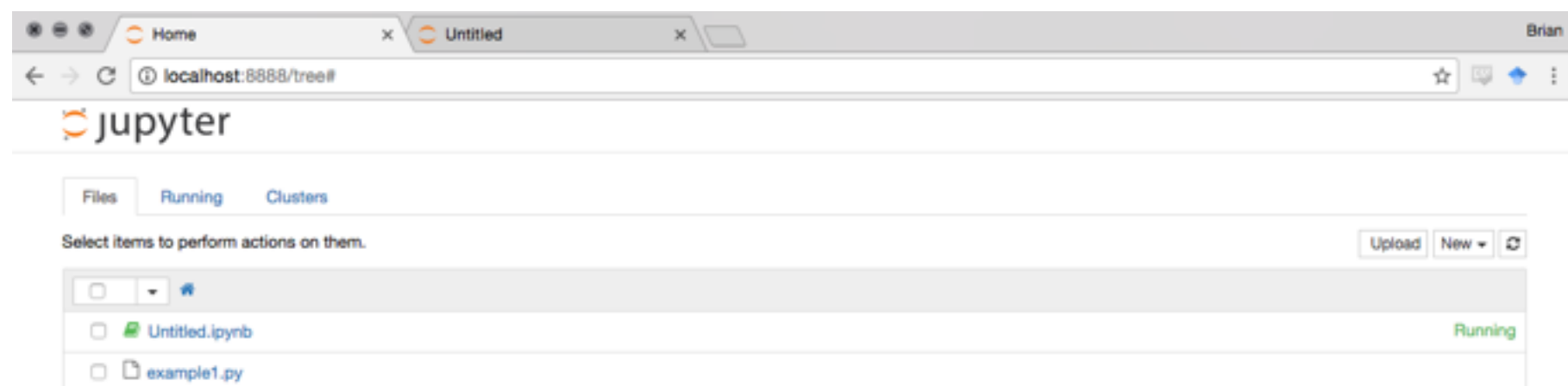


# Interfacing with Python - Jupyter Notebooks

- This did two things....
- Opened a new “Untitled” jupyter notebook in a new tab



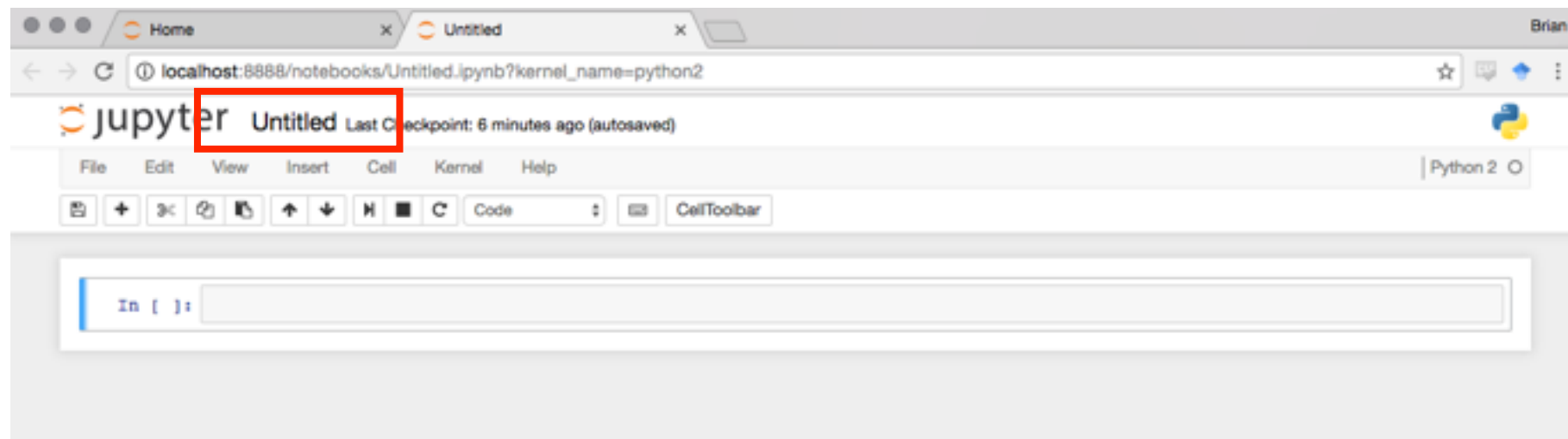
- Added the new “Untitled” notebook to the directory



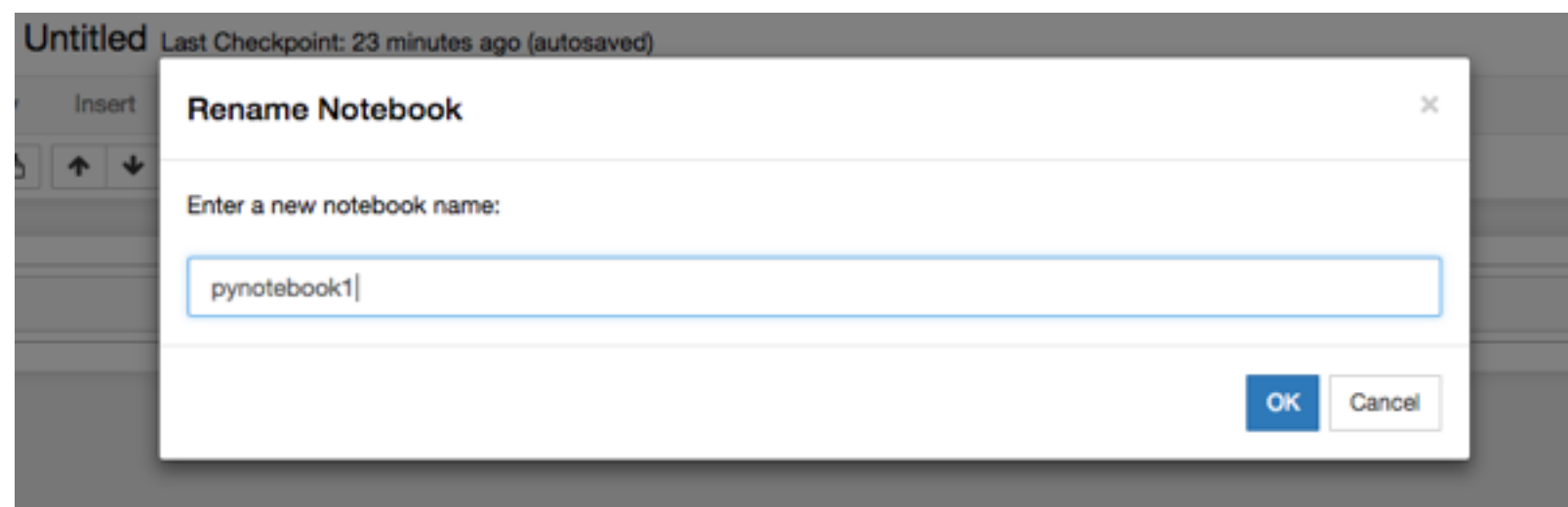


# Interfacing with Python - Jupyter Notebooks

- Let's rename the notebook
- Click the word "Untitled" next to the logo



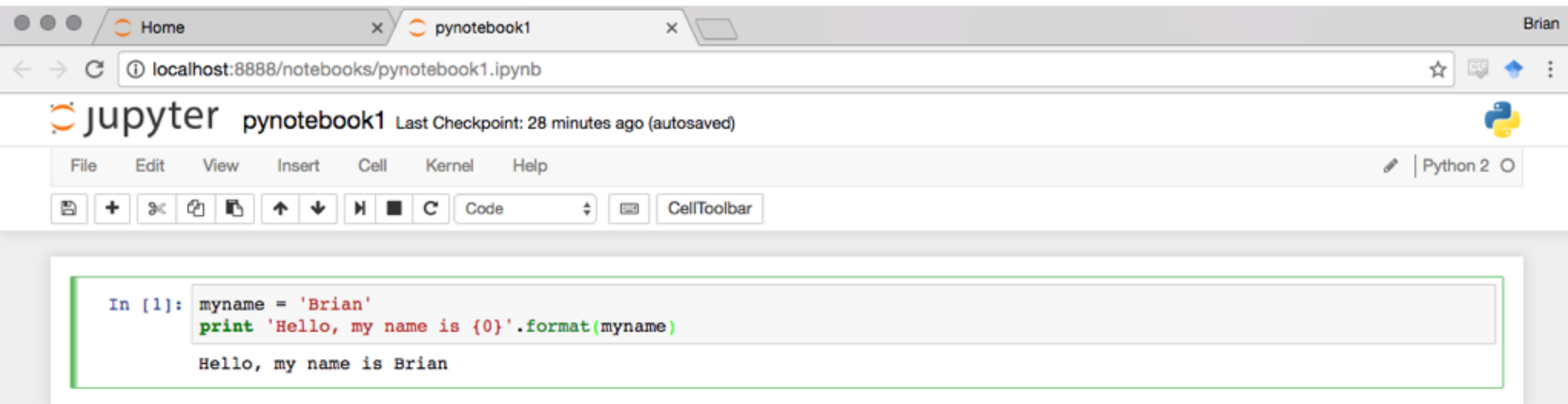
- Change the name to "pynotebook1"





# Interfacing with Python - Jupyter Notebooks

- In the code box type our simple introductory example
  - `myname = 'Brian'`  
`print 'Hello, my name is {0}'.format(myname)`
- To run the code press: `control + return`





# Fundamentals of Jupyter Notebooks

- What happens when you press `control + return`?
  - Essentially the notebook copies that block of code and runs it within an ipython console. (Notice the numbering)
  - It then returns the result below the code block.

A screenshot of a Jupyter Notebook interface in a web browser. The browser tab is titled "pynotebook1" and the address bar shows "localhost:8888/notebooks/pynotebook1.ipynb". The Jupyter logo and "pynotebook1" are visible in the top left, with a status message "Last Checkpoint: 28 minutes ago (autosaved)". A menu bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", and "Help". Below the menu is a toolbar with icons for file operations and cell execution. The main area contains a code cell with the following content:

```
In [1]: myname = 'Brian'
 print 'Hello, my name is {0}'.format(myname)
```

The output of the cell is "Hello, my name is Brian". A red arrow points from the code cell to a terminal window on the right. The terminal window is titled "bvegetabile - python - 122x24" and shows the output of the code execution:

```
Last login: Wed Oct 26 09:31:16 on ttys000
bvegetabile ~$python
Python 2.7.11 |Anaconda 4.0.0 (x86_64)| (default, Dec 6 2015, 18:57:58)
Type "copyright", "credits" or "license()" for more information.

IPython 4.1.2 -- 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]:
```



# Fundamentals of Jupyter Notebooks

- What happens when you press `control + return`?
  - As with an ipython console all of the variables that you entered are available in subsequent lines
  - This can be useful if you only want to run blocks of code at a time.

The screenshot shows a web browser window with the Jupyter Notebook interface. The browser address bar shows `localhost:8888/notebooks/pynotebook1.ipynb`. The Jupyter interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar. A code cell is selected, containing the following Python code:

```
In [1]: myname = 'Brian'
 print 'Hello, my name is {0}'.format(myname)
```

The output of the code cell is displayed below the code:

```
Hello, my name is Brian
```

A red arrow points from the code cell to a terminal window on the right, which shows the IPython shell output:

```
bvegetabile ~$python
Python 2.7.11 |Anaconda 4.0.0 (x86_64)| (default, Dec 6 2015, 18:57:58)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- 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]:
```



# Fundamentals of Jupyter Notebooks

---

- Lots of functionality. See (Help > Keyboard Shortcuts)
- Blocks can be of three types, we outline the main two:
  - Code - This is what a block defers to. Used for entering code which we wish to execute
  - Markdown - Useful for HTML-like write ups. Additionally supports LaTeX commands for mathematical symbols
    - To change to from a code cell to a markdown cell type `esc` then `m`
- The active block will be indicated by a green border

# Fundamentals of Jupyter Notebooks

## - Useful Keyboard Shortcuts

---



- From within the current cell, to run the current cell
  - `control + enter`
- From within the current cell, to run the current cell then select the cell below it (or insert a cell below if none exists)
  - `shift + enter`
- From within the current cell, to adding a cell below the active cell
  - `esc` then `b`
- From within the current cell, to adding a cell above the active cell
  - `esc` then `a`
- From within the current cell, to remove (or cut) the active cell from the worksheet
  - `esc` then `x`
- From within the current cell, to add a cut cell above the current cell
  - `esc` then `shift+v`
- From within the current cell, to add a cut cell below the active cell
  - `esc` then `v`



# Fundamentals of Jupyter Notebooks

## - Further Reading

---



- To fully utilize the power of Jupyter notebooks for reproducible research it is worth investigating
  - Jupyter Documentation :
    - <https://jupyter.readthedocs.io/en/latest/index.html>
  - Markdown: Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML).
    - <http://daringfireball.net/projects/markdown/syntax>
  - LaTeX: LaTeX can be used to within the markdown structure of Jupyter notebooks to integrate visually beautiful HTML with mathematical equations

# Python Overview - Basics of the Python Language

# Primary Topic's Covered Today

---

- Basic Data Types
- Logical Operations
- Collection Data Types
- Control Flow Structures
- Functions
- For a more comprehensive overview
  - <https://docs.python.org/2/tutorial/>

# Basic Data Types

# Keywords and Identifiers

---

- Variables in python are called object references and the names given to them are called *identifiers* or plainly *names*
- The first character of an identifier must be a letter
- Additionally identifiers are case sensitive
- There are also “reserved” keywords which cannot be used as identifiers for variables or objects.

|        |          |         |        |        |        |       |
|--------|----------|---------|--------|--------|--------|-------|
| and    | continue | except  | global | lambda | raise  | yield |
| as     | def      | exec    | if     | not    | return |       |
| assert | del      | finally | import | or     | try    |       |
| break  | elif     | for     | in     | pass   | while  |       |
| class  | else     | from    | is     | print  | with   |       |

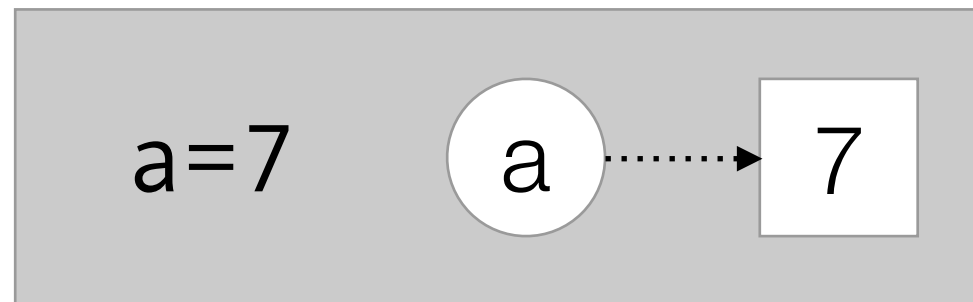
# Keywords and Identifiers - Some Examples

---

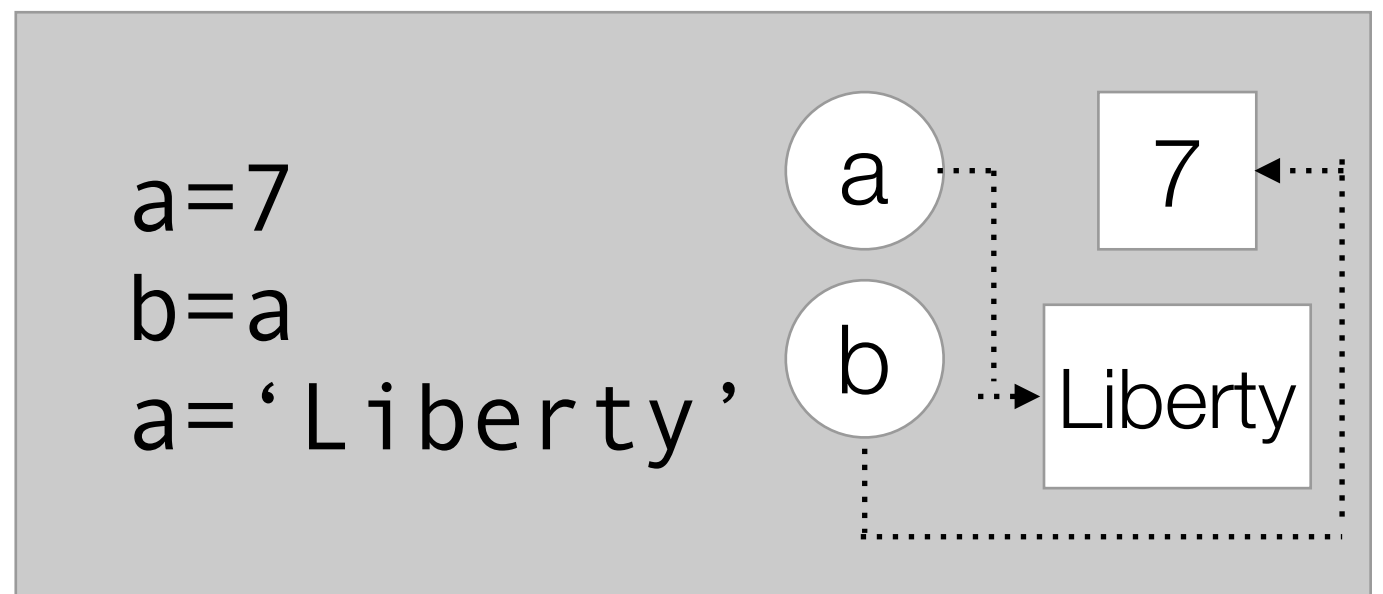
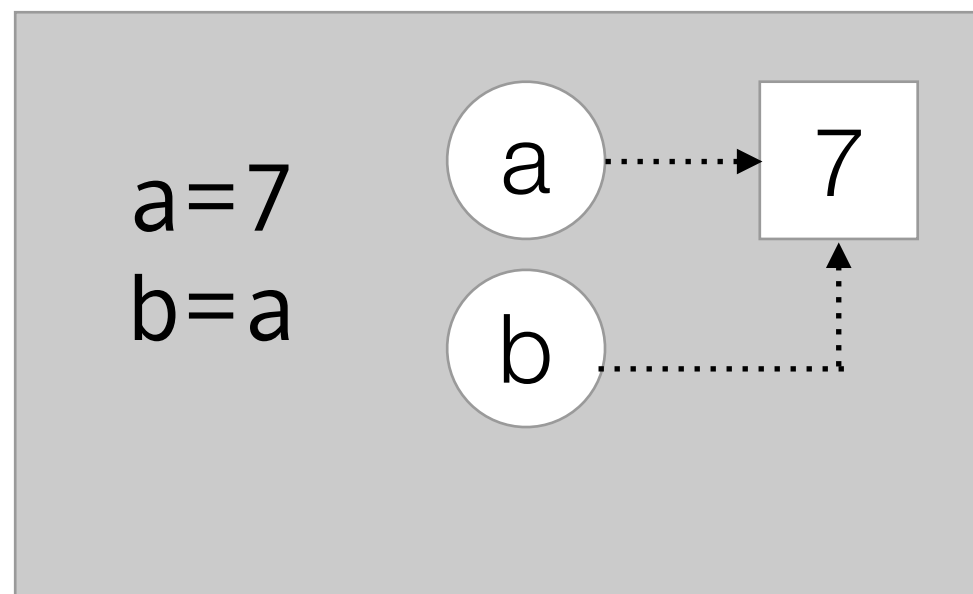
- `myname = "Brian"`  
`todaysweather = 'Sunny'`  
`tomorrowsWeather = 'Cloudy'`  
`Today'sTemp = 77`  
`FAVORITE_color = 'blue'`
- The convention or style for variable names should be consistent (camel case, or using underscores)
- Notice none of these start with a number

# Relationship between objects and object references

---



Circles represent object references  
Rectangles represent objects in memory



**The variable `b` keeps its reference to the value 7**

# Types of Numbers

---

- There are a few major types of numbers that are of basic use in Python
  - **boolean**
  - **integers**
  - **floating-point**
  - complex
  - decimal



# Booleans

---

- The two Boolean objects are `True` and `False`
- Additionally the function `bool()` attempts to convert objects to a boolean object
  - With no arguments it returns `False`
  - With a Boolean object it returns a copy of the original
  - Any other object it attempts to convert to a Boolean
- Using values `1` and `0` instead of `True` and `False` almost always works fine, but the more appropriate syntax is to use Boolean types

# Integers

---

- Integers are of the `int` type and are specified by any number *without* a decimal point
- ```
a = 7          # This is an integer  
b = 7.0        # This is not  
c = 7.         # Neither is this
```
- After boolean numbers, integers are the most basic type
- To attempt to convert an object to an integer type we have the function `int()`

Floating Point Numbers

- There are a few different versions of floating point numbers in Python
 - `float`, `complex`, `Decimal` (from the library `decimal`)
 - The most often used will be the `float` and the `complex` types
 - The `Decimal` type can be used for really high precision operations
- As with `int()` and `bool()`, there is a function `float()` which can be used to create objects of the `float` type or convert objects to the `float` type

Useful Numeric Operations and Functions

Syntax	Description
<code>x + y</code>	Adds number x and number y
<code>x - y</code>	Subtracts y from x
<code>x * y</code>	Multiplies x by y
<code>x / y</code>	Divides x by y
<code>x // y</code>	Floor division of x by y; returns <code>int</code> type
<code>x % y</code>	Modulus operation; remainder of dividing x by y
<code>x**y</code>	x raised to the power y
<code>abs(x)</code>	Takes the absolute value of x
<code>round(x, n)</code>	Rounds x to the n digits.

A quirk of Integer Division

- A feature that almost always trips up those new to Python is the way in which integer division works
- Consider
 - `a = 1 / 3`
- We would expect `a` to return `0.333333333`, but instead it returns `0`
- This is because floor division is performed and another variable of type `int` is returned
- To correct this one of the values needs to be a floating point
 - `b = 1. / 3` or `c = 1.0 / 3` or `d = 1 / 3.` or `e = 1.0 / 3.0`

Even numbers are objects...

- `a = 7`
 `type(a)` # Returns the type of a - int
 `dir(a)` # Lists the methods and attrs
- `b = 4.0`
 `type(b)`
 `dir(b)`
- `c = True`
 `type(c)`
 `dir(c)`

Strings

- One of the most versatile data types in Python is the `str` datatype
 - Objects can be attempted to be converted to a string using the function `str()`.
- Strings hold a sequence of unicode characters
- String literals can be created by enclosing a sequence of characters in either single or double quotes

`myname = "Brian"` or `myname = 'Brian'`

but the notation must be consistent

Strings

- Additionally, Python provides capability for multi-line strings.
- These are useful for using Python to generate flat text files with lots of text without printing line by line.
 - Useful in concert with the `.format()` method
- To create a multi-line string enclose a sequence of characters in triple-quotations

```
multi_str = """Hello, my name is Brian.  
  
This is a multi-line string."""
```


Understanding Sequences - Indexes, Subsequences, Slices and Striding...

- Python has a syntax for taking subsequences of larger sequences
- Python is **zero indexed** and thus the first index location is 0
- Strings are a sequence of characters and are a good place to learn this notation

s[-16]	s[-15]	s[-14]	s[-13]	s[-12]	s[-11]	s[-10]	s[-9]	s[-8]	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
H	e	l	l	o	,		l	'	m		B	r	i	a	n
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]	s[10]	s[11]	s[12]	s[13]	s[14]	s[15]

- Using this notation we can access individual characters of the string

Understanding Sequences - Indexes, Subsequences, Slices and Striding...

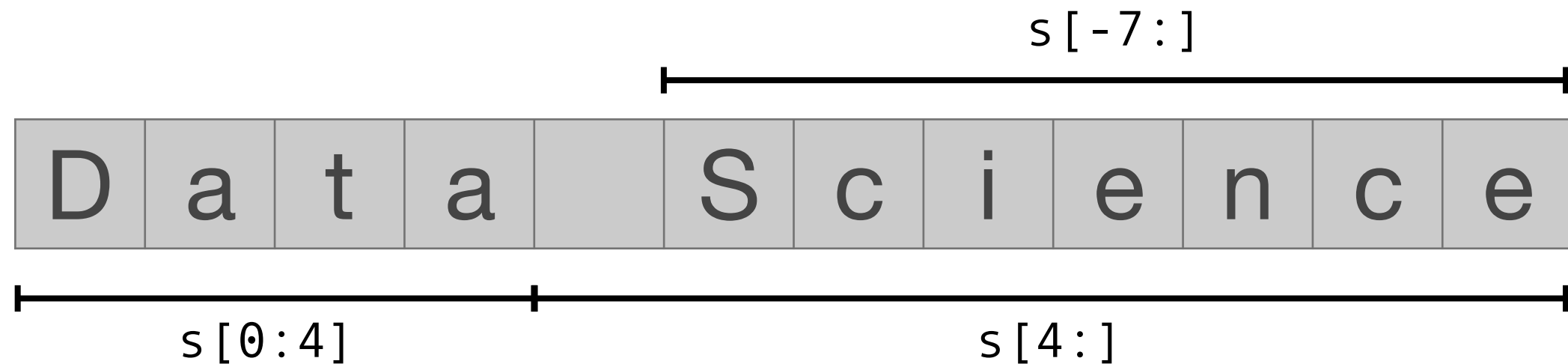
- We may want more than one character though
- A subsequence of a sequence in Python is called a *slice* and has the following notation

```
seq[start]  
seq[start:stop]  
seq[start:stop:step]
```

- Where `start`, `stop`, `step`, must all be integers
- This notation will work with any sequence such as a list, string or a tuple.

Understanding Sequences - Indexes, Subsequences, Slices and Striding...

- Consider the following sequence



- ```
s[0:4] == 'Data'
```

```
s[4:] == 'Science'
```

```
s[-7:] == 'Science'
```

```
s[::2] == 'Dt cec'
```

```
s[::-1] == 'ecneicS ataD'
```

# Methods available to Strings

---

- A major strength of Python is its ability to manipulate strings.
  - Often the primary reason that I give for learning Python in addition to R
- Each `s t r` object carries with it a set of methods (or functions), which allow easy manipulation of the string
- This with logical operations make manipulating strings in Python fairly straight forward
  - That is up until you need to learn regular expressions...

# Methods available to Strings

---

- Below is an overview of some of the methods available.
- As you can see these are very handy when parsing documents and strings

| Syntax                                            | Description                                                                                                                                                    |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>s.capitalize()</code>                       | Returns a copy of string <code>s</code> with the first letter capitalized                                                                                      |
| <code>s.find(t, start, end)</code>                | Returns the leftmost position of <code>t</code> in <code>s</code> (or in the slice <code>s[start:end]</code> ) or <code>-1</code> otherwise                    |
| <code>s.format(...)</code>                        | Returns a copy of <code>s</code> formatted according to the given arguments                                                                                    |
| <code>s.lower()</code> and <code>s.upper()</code> | Returns a lowercase copy of <code>s</code> ; or an uppercase copy of <code>s</code>                                                                            |
| <code>s.replace(t, u, n)</code>                   | Returns a copy of <code>s</code> with every (or a maximum of <code>n</code> if given) occurrences of string <code>t</code> replaced with string <code>u</code> |
| <code>s.split(t, n)</code>                        | Returns a list of strings splitting at most <code>n</code> times on str <code>t</code> .                                                                       |
| <code>s.strip(chars)</code>                       | Returns a copy of <code>s</code> with leading and trailing whitecaps (or the characters in str <code>char</code> ) removed.                                    |
| <code>s.zfill(w)</code>                           | Returns a copy of <code>s</code> , which if shorter than <code>w</code> is padded with leading zeros to make it <code>w</code> characters long.                |

# Further Reading - Completely understanding Strings

- <https://docs.python.org/2/library/string.html>

The screenshot shows a web browser displaying the Python documentation for the `string` module. The browser's address bar shows the URL `https://docs.python.org/2/library/string.html`. The page has a dark blue header with navigation links like "Python", "2.7.12", "Documentation", "The Python Standard Library", and "7. String Services". A sidebar on the left contains a "Table Of Contents" for the `string` module, listing sections like "String constants", "Custom String Formatting", and "Format String Syntax". The main content area is titled "7.1. string — Common string operations" and includes a "Source code: Lib/string.py" link. Below this, a paragraph explains that the `string` module contains constants and classes, and that Python's built-in string classes support sequence type methods. The section "7.1.1. String constants" is highlighted, and it lists constants like `string.ascii_letters` and `string.ascii_lowercase` with their descriptions.

Python Software Foundation [US] | <https://docs.python.org/2/library/string.html>

Python » 2.7.12 » Documentation » The Python Standard Library » 7. String Services »

## 7.1. string — Common string operations

Source code: [Lib/string.py](#)

The `string` module contains a number of useful constants and classes, as well as some deprecated legacy functions that are available as methods on strings. In addition, Python's built-in string classes support the sequence type methods described in the [Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange](#) section, and also the string-specific methods described in the [String Methods](#) section. To output formatted strings use template strings or the `%` operator described in the [String Operations](#) section. Also, see the `re` module for string functions based on regular expressions.

### 7.1.1. String constants ¶

The constants defined in this module are:

`string.ascii_letters`  
The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.ascii_lowercase`  
The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

# Logical Operations

# Logical Operations

---

- A fundamental extension to having data types is comparing them or performing logical operations on them
- I'll provide a brief overview of the three main logical operations you'll want to concern yourself with
  - Comparison Operations
  - Logical Operations
  - Membership Operations



# Comparison Operations

---

- As with most programming languages, Python provides a set of binary comparison operators
  - < less than, <= less than or equal to
  - == equal to, != not equal to
  - > greater than, >= greater than or equal to
- Additionally it provides an identity operation, `is`
  - This references location in memory (is this the same object in memory as another), thus want to use `==` or `!=` to test *values*

# Logical Operations

---

- Additionally Python provides a way to test Boolean statements with `and`, `or`, `not`
- Both `and` and `or` use short-circuit logic and return the operand which determines the result
  - This can be tricky if you're not careful
- What this means is:
  - `5 and 3` returns `3`,
  - `5 or 3` returns `5`,
  - `5 and 0` returns `0`

# Membership Operations

---

- We haven't talked about collection data types yet, but we will introduce another type of logical operation
  - This is the `in` operation
- This operator will return whether an object is *within* a collection
  - Additionally can be used with logical operators (`not in`)
- Slow for large ordered collections, but can be very fast on certain data types such as dictionaries and sets

# Collection Data Types

# Types of Collections

---

- There are three types of collection data types
  - Sequence Types, Set Types, Mapping Types
- Often we will want to collect many of the simple data types that we introduced earlier, or collection collections of variables together!
- We deal with collection types all the time working with data

# Sequence Types

---

- Collections of the Sequence type support the membership operator `in`, the size function `len()`, slices, and are iterable
- When iterated, a sequence will provide the items in their order in the sequence
- The primary sequence types that are of interest are
  - `list`, `str`, `tuple`
  - We've already introduced `str`

# Lists

---

- Lists are one of the most versatile data structures in Python.
- A list object is a list of comma separated values (the items of a list) within square brackets.
- `small_list = [1, 2, 3, 4]`
- While lists can be constructed of many different data types they are often homogeneous

# Lists

---

- Lists support operations like simple concatenation

```
my_list_1 = [1,2,3]
my_list_2 = my_list_1 + [4,5,6]
```

- Additionally they have many methods available to them as well.
  - For those familiar with algorithms they can be used as stacks (last-in, first-out)
  - Or they can be used as a queue (first-in, first-out), but there are more efficient data structures developed for that purpose.
- Are used extensively throughout data science and will be a primary data structure used for manipulating data



# Methods available to lists

| Syntax                         | Description                                                                                                                                                                                                                                                                                                                               |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>list.append(x)</code>    | Add an item to the end of a list.                                                                                                                                                                                                                                                                                                         |
| <code>list.extend(L)</code>    | Extend the list by appending all the items in the given list                                                                                                                                                                                                                                                                              |
| <code>list.insert(i,x)</code>  | Insert an item at a given position. The first argument is the index of the element before which to insert,                                                                                                                                                                                                                                |
| <code>list.remove(x)</code>    | Remove the first item from the list whose value is x. It is an error if there is no such item.                                                                                                                                                                                                                                            |
| <code>list.pop([i])</code>     | Remove the item at the given position in the list, and return it. If no index is specified, <code>a.pop()</code> removes and returns the last item in the list. (The square brackets around the <code>i</code> in the method signature denote that the parameter is optional, not that you should type square brackets at that position.) |
| <code>list.index(x)</code>     | Return the index in the list of the first item whose value is x. It is an error if there is no such item.                                                                                                                                                                                                                                 |
| <code>list.count(x)</code>     | Return the number of times x appears in the list.                                                                                                                                                                                                                                                                                         |
| <code>list.sort(params)</code> | Sort the items of the list in place                                                                                                                                                                                                                                                                                                       |
| <code>list.reverse()</code>    | Reverse the elements of the list, in place.                                                                                                                                                                                                                                                                                               |

# Lists of Lists: Multidimensional Lists

---

- Since a list can contain any object, it naturally follows that they can contain lists

```
multi_list = [[1,2,3], [4,5,6]]
```

- We introduce these because it is worth understanding how to use multiple indices.
- To get to the element 3, use indexing as follows:

```
multi_list[0][2]
```

# Lists of Lists: Multidimensional Lists

---

- Why does this work? The object `multi_list[0]` is itself a list. Namely the list `[1, 2, 3]`.
- To get to the element 3, we can use the same convention with a list

`[1, 2, 3][0]`

which returns 3.

- This chaining together of indices or methods is very useful throughout the Python language and we'll make more use of this later

# Tuples

---

- A tuple consists of a number of values separated by commas (and often times contained in parentheses)
  - ```
my_tuple = (a, b, c)
my_tuple2 = 4, 1, 5
my_tuple3 = 4,          # Ugly but it works
```
- Similar to lists, but used in different situations for different purposes
- Tuples are immutable (unchangeable), and contain a heterogeneous sequence of elements
 - `my_tuple[0] = d` will return an error
 - Items can be accessed via unpacking or indexing
 - Conversely, lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list

Lists are mutable...

Tuples are immutable....

- Consider the following small bit of code

```
a = [1, 2, 3]
b = (1, 2, 3)
a[0] = 4
b[0] = 4
```

- We see that we are able to edit the value at index zero of the list a.
- Conversely when attempting to edit the first index of the tuple we get the following error

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

Tuples - Unpacking

- Tuple packing/unpacking will be referenced and useful throughout learning Python
- Tuples can be packed in the following manner

```
t = 123.0, 'hello', 4
```

and similarly unpacked into new variables

```
a, b, c = t
```

- Here `t` is unpacked and each item is assigned to the identifiers `a`, `b`, `c` at the beginning of the line.

Set Types

- A `set` type is a collection which supports the membership operator `in`, the size function `len()`, and is iterable
- A set is an unordered collection of zero or more object references which refer to objects
 - Since it is unordered, there is no concept of a slice or stride as with strings and tuples
- In the interest of time, we won't go into the details of sets, but they can be incredibly useful for using set operations such as `union`, `disjoint`, `intersection`

Mapping Types

- A mapping object maps hashable values to arbitrary objects
 - An object is hashable if it has a hash value which never changes during its lifetime and can be compared to other objects
 - Hashable objects which compare equal must have the same hash value.
 - A hash value is a numeric value of a fixed length that uniquely identifies data.
 - All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are.
- Mappings are mutable objects
- Currently only one standard mapping type, the dictionary `dict()`

Dictionaries

- A dictionary is a set of **unordered** *key:value* pairs where each *key* is a unique identifier
- Keys can be any immutable type; strings and integers can always be used as keys
- Tuples can be used as keys if they contain only strings, numbers, or tuples; otherwise it cannot
- Dictionaries, lists, and tuples are all arguably the most common data collections that you'll run into in python when manipulating data

Dictionaries

- Examples of dictionaries

```
my_dict = {}          # Empty dictionary
my_dict2 = dict()     # Empty dictionary
my_dict3 = {1:2, 2:3, 3:4}
my_dict4 = {'one':1, 'two':2, 'three':3}
```

- To access items in a dictionary they must be accessed via their *key* names

```
dict[key]
```

```
#Example: my_dict4['three']
```

Methods for Dictionaries

Syntax	Description
<code>d[key]</code>	Return the item of <code>d</code> with key <code>key</code> . Raises a <code>KeyError</code> if <code>key</code> is not in the map.
<code>d[key] == value</code>	Set <code>d[key]</code> to <code>value</code> .
<code>del d[key]</code>	Remove <code>d[key]</code> from <code>d</code> . Raises a <code>KeyError</code> if <code>key</code> is not in the map.
<code>d.clear()</code>	Remove all items from the dictionary.
<code>d.get(key[, default])</code>	Return the value for <code>key</code> if <code>key</code> is in the dictionary, else <code>default</code> . If <code>default</code> is not given, it defaults to <code>None</code> , so that this method never raises a <code>KeyError</code> .
<code>d.items()</code>	Return a copy of the dictionary's list of (key, value) pairs.
<code>d.keys()</code>	Return a copy of the dictionary's list of keys.
<code>d.values()</code>	Return a copy of the dictionary's list of values.
<code>d.update(key)</code>	Update the dictionary with the key/value pairs from <code>other</code> , overwriting existing keys. Return <code>None</code> .

An interesting property of object references of collections

- Consider the following example

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

```
a[1] = 4
```

```
print a, b
```

- Let's investigate this behavior...

An interesting property of object references of collections

- What has happened is that Python has created a **shallow copy** of the variable a
 - This means that b is only a *pointer* to the location of the list of a in memory.
 - When a changes... b does not retain an original copy of the list and is still bound to the location of that list in memory
 - While this is done to conserve memory it means that you have to be careful when creating object references.
- One method (though not universal) of correcting this is the following

```
a = [1, 2, 3]
b = list(a)
```

```
a[1] = 4
```

```
print a, b
```

Control Flow Statements

Control Flow Statements

- Now that we've defined basic data types, operations on these data types and finally collections of simple data types... we can start doing more interesting programming tasks
- To build on this we want to be able to construct the language so that certain statements tell the program how to “flow” between statements

Conditional Branches

- The most simple conditional branch is the `if` statement

```
x = 3
if x == 3:
    print "x = " + str(x)
```

- As you can see, it is of the form

```
if condition:
    #code executed if true
```

- This is why we introduced logical operations before understanding flow statements
- In the above code nothing happens if the statement evaluates to false.

Conditional Branches

- If we want something to happen when the `if` statement fails, we can add an `else` statement.

```
x = 4
if x == 4:
    print 'x was ' + str(x)
else:
    print 'x was not 4'
```

- This allows us to have a condition that will always run when checking a conditional

Conditional Branches

- To add multiple conditions to a program we can nest `elif` statements between the `if` and `else` statements

```
x = 4
if x == 3:
    print 'x was ' + str(x)
elif x == 4:
    print 'x was ' + str(x)
else:
    print 'x not found in conditions'
```

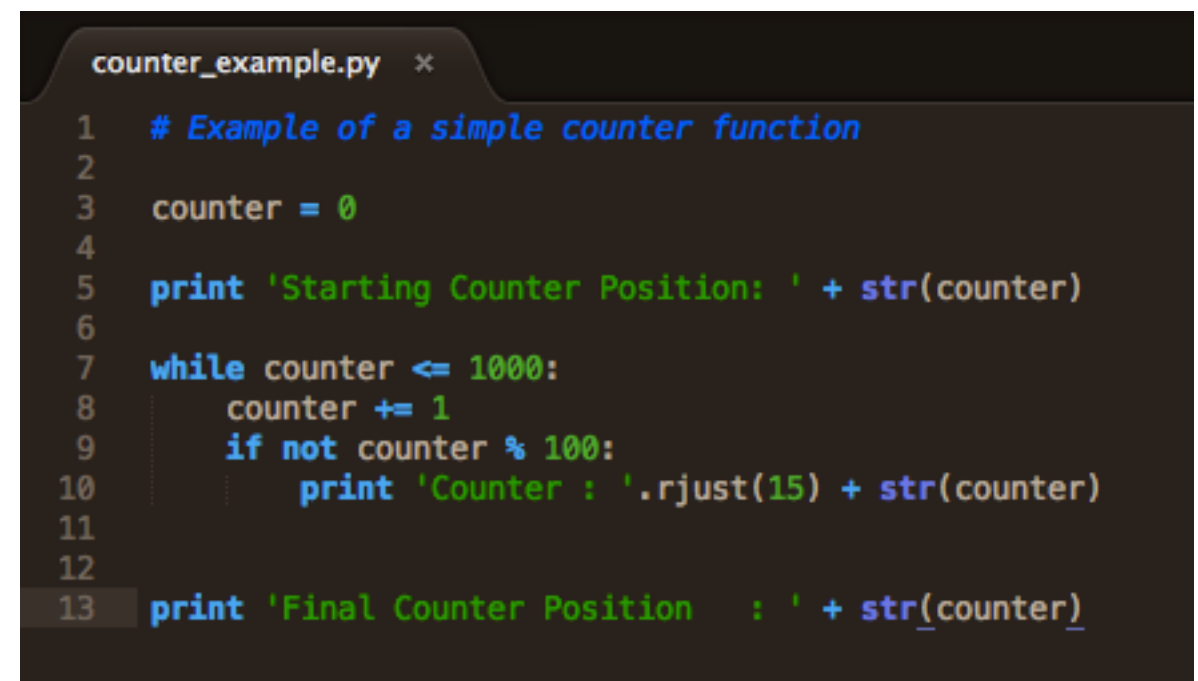
- The first statement must be an `if` statement and the `else` statement does not contain a condition to check.
- As with all python code, the conditions are evaluated in order so understanding the precedence for certain conditions is imperative.

A bit about whitespace and structure

- Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line
 - This in turn is used to determine the grouping of statements.
- The total number of spaces preceding the first non-blank character then determines the line's indentation.
- Python's spacing is what is used to organize the structure of a program.

A bit about whitespace and structure

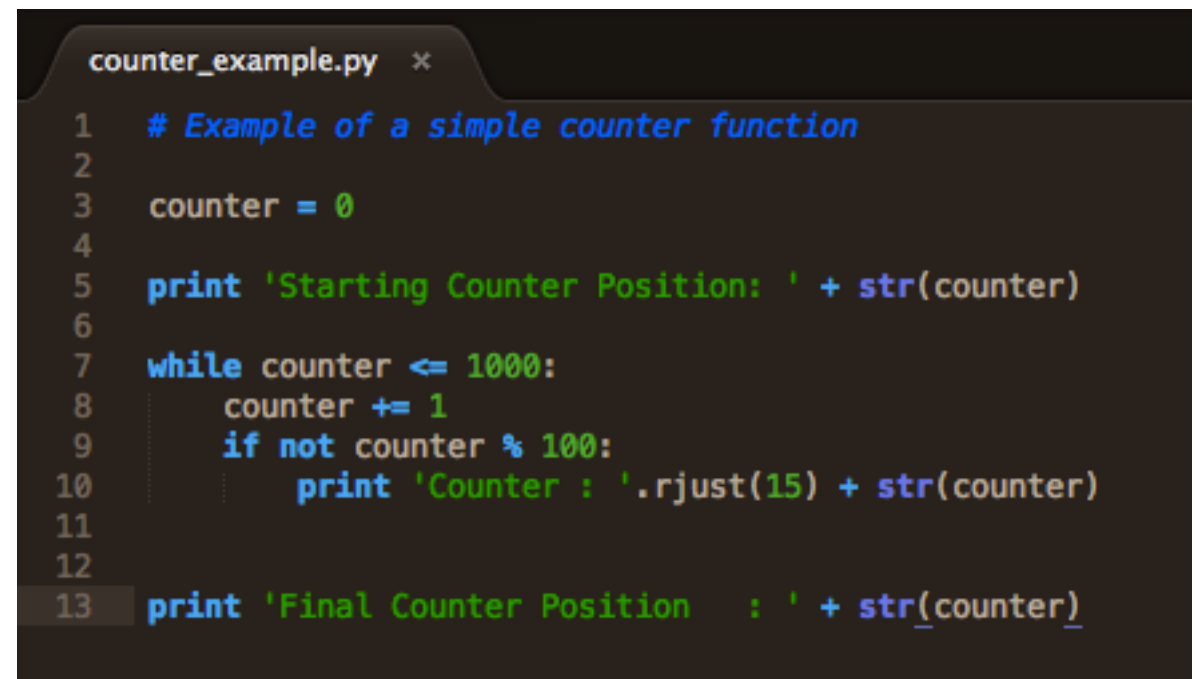
- Organization of blocks of code by whitespace is actually very useful
- We can see to the right that the indentation allows us to clearly see where the while loop begins and where the conditional structure begins on the line below it
- Additionally blocks of code are started by a colon
 - See the `while` loop and the `if` statement



```
counter_example.py x
1  # Example of a simple counter function
2
3  counter = 0
4
5  print 'Starting Counter Position: ' + str(counter)
6
7  while counter <= 1000:
8      counter += 1
9      if not counter % 100:
10         print 'Counter : '.rjust(15) + str(counter)
11
12
13 print 'Final Counter Position  : ' + str(counter)
```

A bit about whitespace and structure

- Indentations can either be groups of space or tabs so long as they are consistent within the document.
- The example to the right uses 4 spaces for indentation.
- Additionally many text editors can be set to show these indents for you!



```
counter_example.py ×
1  # Example of a simple counter function
2
3  counter = 0
4
5  print 'Starting Counter Position: ' + str(counter)
6
7  while counter <= 1000:
8      counter += 1
9      if not counter % 100:
10         print 'Counter : '.rjust(15) + str(counter)
11
12
13 print 'Final Counter Position  : ' + str(counter)
```

A bit about whitespace and structure

- The code to the right highlights a 'bad indent'
- The if block is no longer properly and consistently indented
- Attempting to run this code will throw an `IndentationError`

```
counter_example.py x
1  # Example of a simple counter function
2
3  counter = 0
4
5  print 'Starting Counter Position: ' + str(counter)
6
7  while counter <= 1000:
8      counter += 1
9      if not counter % 100:
10         print 'Counter : '.rjust(15) + str(counter)
11         print 'Bad Indents'
12
13
14  print 'Final Counter Position  : ' + str(counter)
```

Tabs vs. Spaces

- So do you use tabs or space?

This has been source of debate

(Click for YouTube clip)

- From the PEP 8 Python Style Guide:
 - Spaces are the preferred indentation method.
 - Tabs should be used solely to remain consistent with code that is already indented with tabs

Loops

- Another thing that is often important when controlling the flow of a program is to iterate or loop over the objects in a collection
- The two primary looping statements are the `while` and `for` loops.
- In addition to the `for` loop, there are functions built into Python which are often used with the `for` loop. Namely the `range()` and `enumerate()` functions

While Loops

- A simple description of a while loop is that is used to execute some code zero or more times
 - The number of times that the code is executed is determined by a boolean condition.
- While loops are often useful when the structure of the object is unknown or if we don't know how many times we will need to do something.
 - Caution: You will create infinite loops at some point....
- The simple example is as follows:

```
x = 0
while x < 10:
    x += 1
print x
```

While Loops

- Useful with while statements are the commands `break` and `continue`
- These commands are often used within conditional branches to change or divert the flow of a loop

```
x = 0
while True:
    if x == 10:
        break
    x += 1
print x
```

- The above code snip does the same as the previous slide.

While Loops

- Similarly the continue command can be used to skip certain steps.
- The following code prints all even numbers to 10:

```
x = 0
while x <= 10:
    x += 1
    if x % 2:
        continue
    print x
```

For Loops

- Python's `for` loop is used to iterate through the objects of an object which is *iterable*.
 - An iterable is any object which can be iterated over, including strings
- For loops will be used fairly often within data science, because we often had a set of observations with which we want to iterate over.
- Similar to the `while` loop, the `break` and `continue` statements are available to the `for` loop for controlling the flow through the iterable object

For Loops

- Reuses the `in` statement from membership operations
- Syntax

```
for variable in iterable:  
    #some code here
```

Example

```
x = ['one', 'two', 'three']  
for number in x:  
    print number
```

For Loops

- Two commands will be very useful in conjunction with for loops: `range()` and `enumerate()`.
- `range()` creates sequence of integers

Syntax `range(stop)`
 `range(start, stop[, step])`

Example `range(10)` # Outputs a list of
 # 10 ints (0 to 9)
 `range(0, 10, 2)` # Outputs a list
 # of even numbers

- This is useful if you know the number of objects and want to create a list of indices to iterate through.

```
x = [1,2,3]
for ind in range[3]:
    print x[ind]
```

For Loops

- The `enumerate()` command returns an iterable object that ties each object to its position in the iteration process.
- To understand how this works it is best to see it in action

```
a, b, c = 15.0, 33.0, 3.14
x = [a, b, c]
for ind, obj in enumerate(x):
    print ind, obj
```

- The `enumerate()` function is useful when the order of two lists is useful and comparable.

```
my_dict = {'one':1, 'two':2, 'three':3}
for ind, obj in enumerate(my_dict):
    print ind, obj
```

Recall that dictionaries have an arbitrary ordering, therefore it may not make sense in this use case!

Creating Functions

Built-in functions

- Python has many, many, many built-in functions already at your disposal.
- These functions are always available

<https://docs.python.org/2/library/functions.html>

- They don't encompass everything that could possibly be done with the language and therefore it is useful to know how to create your own functions

Defining Functions

- The keyword `def` introduces a new function definition
- It must be followed by the function name and the parenthesized list of formal parameters
- The statements that form the body of the function start at the next line, and must be indented.

```
def function_name(parameters):  
    pass
```

Example function

- ```
def fib(n): # write Fibonacci series up to n
 """Print a Fibonacci series up to n."""
 a, b = 0, 1
 while a <= n:
 print a,
 a, b = b, a+b

fib(1000) # This line calls the function
```

# Defining Functions - Introducing default values

---

- Functions can be specified with default values, so that the function can take in less arguments than are specified
  - The default values are then used when specified
- ```
def print_names(x = ['Brian']):  
    if len(x) == 1:  
        print 'There was one name: ' + x[0]  
    else:  
        print 'There were {} names'.format(len(x))  
        for ind, name in enumerate(x, start=1):  
            print 'Name {0}: {1}'.format(ind, name)  
  
print_names()  
print_names(['Brian', 'Steve', 'Michael'])  
print_names('Brian')    # Don't forget strings are iterable
```

Returning values from a Function

- Most of the time you will need to return something from a function.
- This can be accomplished with the return statement at the end of the function

```
def function_name(parameters):  
    # Some operations  
    return object
```

Returning values from a Function

- Example of how to return multiple numbers

```
def min_max(x):  
    return min(x), max(x)
```

```
some_numbers = [1, 5, 15, -1, 6, 3.]  
print min_max(some_numbers) # a tuple  
min_x, max_x = min_max(some_numbers)
```

The function `min_max()` returns a tuple containing the minimum and maximum of the object `x`.

This can be unpacked into multiple variables using the unpacking notation that was described earlier in the workshop!

It's impossible to teach everything...

- Clearly I don't have enough time to teach everything today, but hopefully I've included enough of the fundamentals for you to get started.
- Python has many quirks that take getting used to, but in my opinion it is worth learning to for all the benefits of the language.

Consider the output from following code

```
x = [0] * 10
y = [x] * 4
y[1][2] = 1
print y
```

At first this will seem unintuitive because this changes ALL rows as opposed to the first row. Understanding object references further would make this behavior clear.

- To really understand the language explore the Python documentation!

Break for Simple Exercises

Libraries in Python

- Introducing *Numpy*, *Pandas*, *Sci-kit Learn*,
Statsmodels

Enhancing Python with Libraries

- Out of the box there are many functions available to you in Python.
- But you are not limited to the ‘out of the box’ version of Python and there are entire libraries of functions that are available to use within Python
- One of the major reasons for installing the Anaconda distribution of Python is that it is a package manager that helps keep all of these libraries (and more) managed

```
conda update conda
```

Python's Standard Library

- Python's standard library is very extensive
- The standard library provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers
- Additionally it provides standardized solutions to many problems that occur in programming
 - i.e. Often in Python it's worth checking to see if someone has done it before you reinvent the wheel.

Common Modules, Packages, Solutions

- **String Services**

`string` - Common string operations
`re` - Regular expression operations

- **Data Types**

`datetime` - Basic date and time types
`collections` - High-performance container datatypes
`array` - Efficient arrays of numeric values

- **Numeric and Mathematical Modules**

`math` - Mathematical functions
`decimal` - Decimal fixed point and floating point arithmetic
`random` - Generate pseudo-random numbers

- **File and Directory Access**

`glob` - Unix style pathname pattern expansion

- **File Formats**

`csv` - CSV File Reading and Writing

- **Generic Operating System Services**

`os` - Miscellaneous operating system interfaces

- **Python Runtime Services**

`sys` - System-specific parameters and functions

- **Many, many, many more**

<https://docs.python.org/2/library/>

Using external modules and libraries

- It's nice that all of these modules and libraries exist, but it isn't apparent yet how to interface with them
- Python has a syntax for imports, for example

```
import math  
print math.pi
```

- The above makes available the module math. The second line prints the attribute pi from the math module
- To see all of the functions available to the math module use tab completion on math.<tab> in the Jupyter notebook

Using external modules and libraries

- There is an alternative way of importing modules and packages where we shorten their names
- This is especially useful because using the 'dot'-notation of the python objects can require a lot of typing and programmers want to minimize their typing as much as possible
- This motivates the `as` statement

```
import math as m  
print m.pi
```

- Many packages have specific short hand names that are generally agreed upon in the developer community (for example the package `numpy` is generally brought in as `np`)

Importing specific functions

- Sometimes we may only want to import specific functions from a module. This is possible too to save resources
- As an example suppose we only wanted the value of pi from the math module

```
from math import pi  
print pi
```

- Now we can access the value of pi without typing math first.
- We can also use the `as` statement here, for example

```
from math import pi as p
```

- *Note: that while we **can** do this such simple variable names as `p` are unadvised since more verbose code is arguably more readable and there is less of a chance of creating a conflict with specific names.*

Importing specific functions

- If we wanted to be really aggressive we could import all of the functions and attributes from a library with the following notation

```
from math import *
```

- Generally this is unadvisable! **(Don't do it)**

- Consider

```
pi = 5  
from math import pi  
print pi
```

- We see that pi has been replaced by the value from the math module. We're getting into the types of programming that makes us dangerous if we're not careful.

Libraries for Data Science

Package Name	Description
<code>numpy</code>	NumPy is the fundamental package for scientific computing with Python.
<code>scipy</code>	SciPy is an open source Python library used for scientific computing and technical computing.
<code>pandas</code>	The Python Data Analysis Library - pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
<code>sklearn</code>	Scikit-learn is a machine learning library for the Python programming language. It features various classification, regression and clustering algorithms and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.
<code>statsmodels</code>	Statsmodels is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests.
<code>matplotlib</code>	matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
<code>seaborn</code>	Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics.

Getting started with NumPy

- NumPy is imported into python as the identifier np

```
import numpy as np
```

- This is the first package to get familiar with for working with data
- Provides data structures for creating arrays and matrices and for manipulating them

NumPy arrays

- NumPy arrays are one of the reasons for the packages success

```
my_array = np.array([1,2,3.])
```

- They can also be used to create multidimensional arrays

```
matrix_array = np.array([[1,2],[3,4]])
```

- They look awfully familiar to lists...

NumPy arrays vs. Lists

- A simple check of the methods available to the two objects shows the major advantages of lists vs. numpy arrays

```
a = [[1,2], [3,4]]  
b = np.array([[1,2], [3,4]])  
print dir(a)  
print dir(b)
```

Some Methods and Attributes for Arrays

- Here are a sample of some useful methods and attributes of arrays

Syntax	Description
<code>array.shape</code>	Tuple of array dimensions.
<code>array.size</code>	Number of elements in the array.
<code>array.T</code>	Transpose of the array
<code>array.max(axis)</code>	Returns the maximum value of the array along a given axis
<code>array.round(digits)</code>	Return an array with each element rounded to the given number of decimals.
<code>array.sum(axis)</code>	Return the sum of the array elements over the given axis.
<code>array.cumsum(axis)</code>	Return the cumulative sum across a given axis
<code>array.mean(axis)</code>	Return the mean along a given axis
<code>array.var(axis)</code>	Return the variance along a given axis

But numpy has a Matrix Object?

- Numpy provides, in addition to `np.array()`, an additional `np.matrix()` type that you may see used in some existing code. Which one to use?
 - **Short Answer? Use `np.array()`**
- NumPy provides the matrix class for matrix algebra, but you can almost consider it a "subclass" in numpy.
- Additionally, arrays are the standard vector/matrix/tensor type of numpy. Many numpy functions return arrays, not matrices.
- Any algebra you would want to use on a matrix object is available to an array object

Matrix Operations in numpy

- Matrix multiplication can be done using the `.dot()` method

```
A = np.array([[1, 2],  
              [3, 4]])
```

```
b = np.array([1, 2])
```

```
print np.dot(A, b.T)
```

```
# Matrix Mult
```

```
print np.dot(A, b)
```

```
# numpy fixes dim
```

```
print A.dot(b)
```

```
# Chained Method
```

```
print A * b
```

```
# Element-wise
```

Linear Algebra Operation - numpy.linalg

- There are other linear algebra functions within `numpy.linalg`

```
A = np.array([[1, 2],  
              [3, 4]])
```

```
b = np.array([1, 2])
```

```
A_inv = np.linalg.inv(A) #inverse
```

```
A_det = np.linalg.det(A) #determinant
```

```
A_svd = np.linalg.svd(A) #sing.val.decomp
```


Getting Started with Pandas

- Pandas builds on the functionality of numpy, but introduces data frames

```
import pandas as pd
```

- DataFrames are a two-dimensional tabular data structure (very similar to data frames in R)
 - Here the data is actually composed of a numpy array! Therefore much of what we just learned is still useful
 - Named columns for easy access
- We now have almost everything we had with numpy arrays, but with more added functionality

A few ways to create a DataFrame

```
• fake_data = np.array([[ 'Brian', 30],  
                        [ 'Brett', 29],  
                        [ 'Jake', 26],  
                        [ 'Bob', 57]])  
  
data_set = pd.DataFrame(fake_data,  
                        columns=[ 'Name', 'Age'])  
  
print data_set.Name  
print data_set.Age
```

A few ways to create a DataFrame

- ```
fake_data = {'Names': ['Brian', 'Brett',
 'Jake', 'Bob'],
 'Age': [30, 29, 26, 57]}
```

```
data_set = pd.DataFrame(fake_data)
```

```
print data_set.Names
print data_set.Age
```

# Reading Files with Pandas is Easy

---

- Another reason for using pandas is the ease with which it brings data into python

```
pd.read_csv(some_file)
```

- This function can take in many arguments to customize its behavior, but it generally works well without any modifications.