Table of Contents

Preface

This GitBook was written by David Backus, Sarah Beckett-Hile, Chase Coleman, and Spencer Lyon for a course at NYU's Stern School of Business. The idea is to give students experience with economic and financial data and introduce programming newbies to the benefits of moving beyond Excel. We use the Python programming language, specifically Python's data management and graphics tools. If that doesn't whet your appetite, we have a more elaborate sales pitch.

We designed the book to accompany a lie class. We've tried to make it self-contained, but the written word is a poor substitute for the interaction you get in a classroom.

The book comes in multiple formats. You can access it on the internet. Or you can download (and print) a pdf file. The former comes with links, which we think is a huge advantage, and can be updated quickly, but if you like paper by all means try the pdf. All of them are availe at

https://www.gitbook.com/book/nyudatabootcamp/data-bootcamp/details

Related course materials are available at

http://nyu.data-bootcamp.com/

We welcome suggestions. Send them to Chase Coleman or Spencer Lyon. Or, even better, post an issue on our GitHub repository.

Warning

This is **work in progress**. We've written seven chapters so far, more are on the way.

Acknowledgements

This project was Glenn Okun's idea. He really should have done it himself, but we thank him for the idea and his ongoing support. Paul Backus, Hersh Iyer (MBA17), Matt McKay, Kim Ruhl, and Itamar Snir (MBA17) contributed technical support and applications. Ian Stewart provided his usual expert advice on teaching methods. You may also notice a family resemblance to Tom Sargent and John Stachurski's Quantitative Economics, a Python- and Julia-based course in dynamic macroeconomic theory. We thank them for their advice and encouragement.

This is a test of my ability to manipulate the book. Mike Waugh

License

Where are we headed?

Overview. Data management skills are enormously valuable in the modern world. We're going to give you those skills, show you how to apply them to economic and financial data, and maybe tell a few bad jokes along the way. Join us!

Buzzwords. Python, code, Google fu.

This book -- and the course we developed it for -- is about **data**. It's also about **tools** for working with data, which in this case means **Python** and its data-related tools. Our focus is economic and financial data, which is what we know best, but the same tools can be applied to any data. By the end of the course, you will have a better idea where to find data that's useful to you, and you will have command over tools you can use to do something interesting with it. We think your life will be more interesting, too, but maybe that's just us.

Answers to common questions

Why should I do this? It's an investment in your future. You will learn how to process data and communicate its content effectively and efficiently. You will have more fun. And you will be more valuable to current and future employers.

Can't I do what I need in Excel? Excel is a great program, but once you have a little programming experience it will remind you of doing arithmetic on your fingers. With Python, you will be able do routine tasks more efficiently ("automate the boring stuff," as one guide suggests), handle larger data sets, rearrange datasets at will, and generally do things that spreadsheet programs can't do.

What are the prerequisites? There are none. We start at the very beginning and go from there. What you will need is the **courage** to take on a challenge and the **patience** to debug programs that don’t quite work -- and they never work the first time, and often not the second or third time either. Don't panic. Ask for help and remind yourself that patience is a virtue.

What if my quant skills are weak or nonexistent? Then this is the course for you! We do our best to make the material accessible. We're looking beyond quants to marketing, management, and humanities majors. One of our design team was an English major.

Will this turn me into a programmer? You will come out of the course somewhere between Brad Pitt and Jonah Hill in "Moneyball," with a solid foundation for dealing with whatever data comes your way. You will not be ready for a career as a programmer, but you will be able to work effectively with people who know more and be able to do things yourself that Excel users can only dream about.

Will this turn me into a data scientist? Sadly, no. But you will have a solid foundation for pursuing the many technical topics that fall under the rubrics of data science and machine learning. See, for example, the extensive collection of courses on **business analytics** and **data science** offered by our IOMS and CS groups.

Should I take this course if I already know how to code? You're welcome to, and you will learn a lot about data and the data components of Python. But please don't scare the other students.

Is there anything Python can't do? Well, it can't swallow a porcupine. Someone is working on pretty much everything else.

Why data?

We're living in a world of data: data about the economy, data about financial markets, data about your business. Data doesn't solve all our problems, but it's a valuable input to better decisions. For example, how to choose a college major.

Many of our former students tell us that data skills keep them in business. One of our alums analyzes television viewer data for a network. The datasets are too large for Excel, so he uses Python. Another manages attendence data for a major league baseball team. A third works for a quantitative hedge fund, where Python is the tool of choice. A fourth is worried that you won't need him after this course. Even students with non-technical backgrounds tell us that basic data and programming skills are, if not required, at least very useful in their jobs. One of our marketing majors, for example, needs to interface with her company's SQL database to get the data she needs to do her job.

Why Python?

Python is a popular general-purpose programming language that has been used for a broad range of applications. Google uses it. So do Instagram and Netflix. Dropbox is written in Python.

We think Python is the language of choice right now if you want a user-friendly introduction to programming and a useful tool for day-to-day data work. It's a high-level language, which means the language does a lot of the work. It has a broad range of applications and an enormous community of users. You'll come to appreciate both. And it's free and open source. Free means you pay nothing. Open source means you can look at the code if you want to see how something works.

Everyone likes Python

Python isn't just a useful language, it's one people like. We're talking about programmers here, for the most part, but even us novices find that its casual simplicity makes coding fun.

Writer and programmer Paul Ford puts it this way:

> People love [Python] and want it to work everywhere and do everything. They've spent tens of thousands of hours making that possible and then given the fruit of their labor away. That's a powerful indicator. A huge amount of effort has gone into making Python practical as well as pleasurable to use.

He's alluding here to the vast community of users who are developing tools that allow Python to do all kinds of things. Python's data tools are an example: they're not part of the core langauage, they're add-ons developed by users. He adds: "Python people are pretty cool," so there's that, too.

Our approach

There is a method to our madness. We will start quickly so that we can reach more interesting topics quickly -- On the bright side, this means that our workload will be heaviest when the workload of your other classes is lightest. We promise that it gets easier in spite of being a bit tough up front. These are some of our guiding principles to put what is being taught into perspective.

One step at a time. You can't do this in a day. It pays to work on the fundamentals before moving on to advanced data tools. Our friend Tom Sargent preaches, "Don't skip steps." We try to follow this principal.

Stress the basics, ignore the rest. We think once you understand the basics, you'll be in a good position to work out special cases on your own. That allows us to strip out a bunch of confusing detail, which we think is good for everyone.

Learn to teach yourself. The best way to learn new things is to teach yourself: Google your problem and find the resources you need. We build that attitude in from the start, suggesting ways in which you can solve problems yourself. But remember: it also helps to be in a supportive environment, where you can ask for help when you need it.

Online book preferred. We sometimes print out the pdf ourselves, but the online version comes with live links. We'll update it frequently as new ideas come to mind. We think it's a superior user experience.

Code and applications. We attack data applications and programming together. After covering Python basics, we generally organize things around specific applications, covering the relevant aspects of Python along the way. We think it helps to have a context for what we're learning, but the downside is that it's somewhat harder to use the book as a programming reference. We still think it makes sense. Our goal isn't to produce programmers, but people who know enough about programming to offer an interesting perspective on data (often through a graphic).

Work habits

There are no shortcuts in learning how to code. You simply need to spend hours doing it. Progress will seem slow at first, but if you stick with it things will start to look familiar, and even make sense. You may even start to think of projects as fun, and revel in your new-found power over data.

As you work your way up the learning curve, keep this advice in mind:

Don't panic. Learning a new language takes time, it won't happen in a week.

Stick with it. The secret is to keep working. Trust us, things will start to make sense in a couple weeks. Here's a great example. How can you not love someone who writes: "How I learned to stop worrying and love the code"?

Practice, practice, practice. Any time you have something to do with data, try it out in Python. Play around, try new things, have fun. As you gain experience, you'll find that Python becomes easier.

Ask for help. If you get stuck, ask for help -- from friends, from your Bootcamp classmates (open a discussion on our google group), or from us (the teachers of the course). We love this one: "I failed my first computer class miserably. ... The next time something clicked -- I made the decision to raise my hand in class and admit publicly that I was completely lost. To my surprise, I found that not only the teacher, but also other students in class were eager to help."

Make friends. Coding is hard to learn on your own. A second pair of eyes is indespensible. So work with friends, and make new friends who know how to code. Intense coding sessions are a great way to develop relationships.

Work on your Google fu. With a little help from Google, you will find that many of your questions have been asked before. Even better, they have been answered. One way to find them: Google something like "python [problem]." Don't forget the problem; without it, you get pages and pages of snakes.

Be patient. We know, it's easier to say than do, but it pays to take your time. Coding in a hurry is a recipe for frustration and failure.

Wordplay

Python is named for Monty Python, a group of comedians whose humor appeals to the tech crowd. Idle, a well-know Python editor, is a reference to Python-member Eric Idle. The Python Package Index, a repository of Python packages, is commonly known as the Cheese Shop, a reference to a famous Monty Python skit. The Anaconda distribution (next chapter) is a play on the word python.

Resources

The resources section at the end of each chapter is a collection of (mostly) links to things we've found useful. They're more than you need, but give you some recommended options if you want to follow up on a specific topic.

Here we'll say simply that all of the materials for this book and the associated course are posted online:

- Website. Everything is posted on our class website.
- Book. It's hosted by GitBook.
- Code. We give links to the relevant code at the start of each chapter, but if you want them all, look in the Code directory of the GitHub repo. If you save them, **remember to click on the Raw button** in the upper right. (This is an oddity of GitHub, which distinguishes between a display of the file and the file iself.)
- Other materials. Pretty much everything else is available within a repository in our GitHub organization.

Installing Python

Overview. We install Python, then run a test program to make sure it's working.

Python tools. Anaconda distribution, Spyder and IPython/Jupyter coding environments.

Buzzwords. Package, distribution, environment, mtwn.

Code. Python | IPython notebook.

Python is a programming language. Like other computer languages, it comes with an ecosystem of related components. We refer to collections of components as **distributions.** We're going to use a specific distribution -- **Anaconda** -- that comes with lots of components in one user-friendly download.

One kind of component is a user-interface or **environment** for writing and executing code. Here's an analogy: Word and Google docs are "environments" to produce text documents. Both work. We can use either one. The same is true of Python environments; we use the one we find most convenient. Think to yourself: An environment is analogous to Microsoft Word and a Python program is analogous to a Word document.

We'll use two Python environments in this class:

- **Spyder** is a graphical interface that includes an editor, a button to run code, and windows for experimenting and checking documentation.
- **Jupyter** is a browser-based interface for running **Jupyter notebooks**, which combine code, output, and documentation.

We will write and run Python programs in both environments.

This is a lot of jargon to swallow at one time. Don't panic, it will become familiar with use. And anything we don't use you can safely ignore.

Installing Anaconda

Follow these instructions. By which we mean: **follow these instructions exactly!** Creativity is a wonderful thing, but here it will cost you dearly.

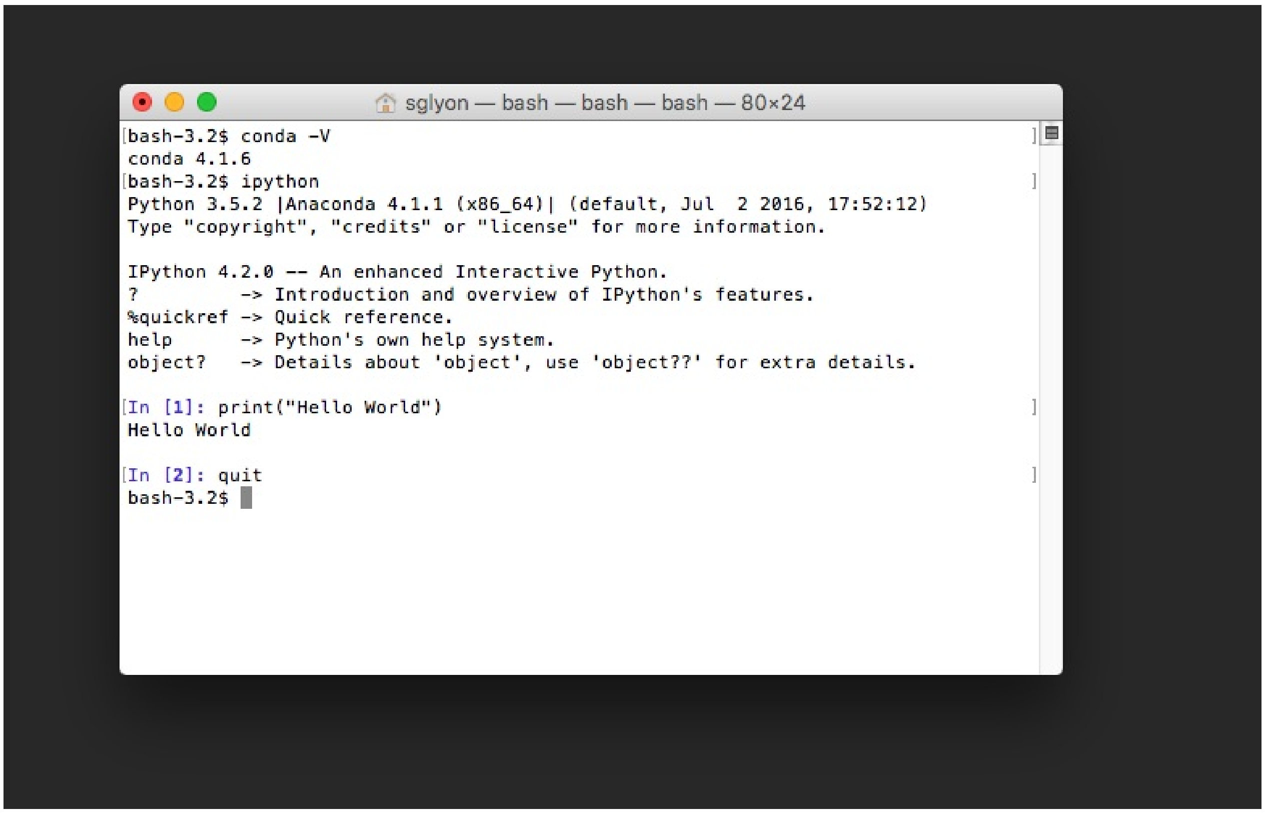
Step 1. Download the Anaconda installer.

- Click HERE or Google "Anaconda downloads."
- Scroll to find "Anaconda for Windows" or further down for "Anaconda for OS X."
- Find the option for **Python 3.5. NOT** Python 2.7! If you get 2.7, you'll have to start over.
- Click the **64-bit Graphical Installer** to start the download.

Step 2. Run the installer. Click on the Anaconda installer you just downloaded to install the Anaconda distribution of Python. Do what it says (keep default options and make sure that **it sets Anaconda as your default Python installation!**).

Step 3. Find and run IPython. Look wherever programs are on your computer.

- Windows: Open the start menu by clicking the start button in the bottom left of your screen or by pressing the windows key. In the search box type "cmd" to open the command prompt. Once you have the command prompt open, type `conda -V`. It should tell you 4.x.y where x and y are just some numbers. Now type `ipython` to open a python session and type `print("Hello World")` and then `quit`
- Mac: open spotlight either by clicking the magnifying glass in the top right of your screen or pressing command and the space bar at the same time. Type “Terminal” and press enter. Once you have the terminal prompt open, type `conda -V`. It should tell you 4.x.y where x and y are just some numbers. Now type `ipython` to open a python session and type `print("Hello World")` and then `quit`



```
bash-3.2$ conda -V
conda 4.1.6
bash-3.2$ ipython
Python 3.5.2 |Anaconda 4.1.1 (x86_64)| (default, Jul  2 2016, 17:52:12)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- 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]: print("Hello World")
Hello World

In [2]: quit
bash-3.2$ 
```

If these commands worked and displayed something similar (version may be slightly different), you now have Anaconda installed and ready to run. Congratulations!

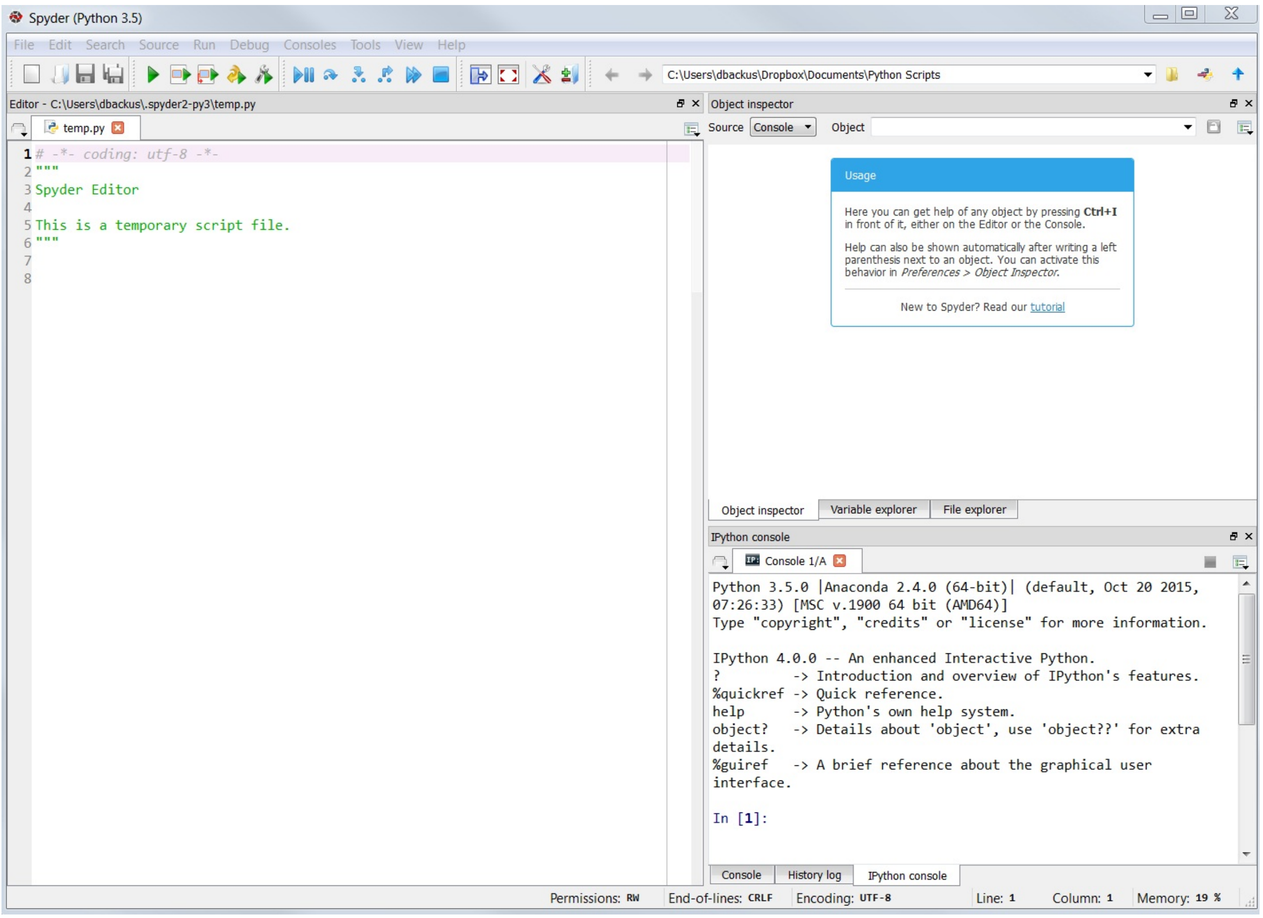
Coding environments

Coding environments are pieces of software we use to write and run code. The best ones make coding easy, even pleasurable, strange as that might sound. We'll use two: **Spyder** and **Jupyter**. We will access both of these through either the command prompt (Windows) or terminal (Mac) which can be opened as described above. In the steps that follow I will refer to "terminal" -- This is what it is called on a Mac, but, if you are on a Windows machine, please just replace the word terminal with the word command prompt everywhere in this section.

If you still have a terminal open, great. If not, please open a new one (described in Step 3 above)

Spyder. Spyder is a graphical environment with an editor for writing programs, a console for trying out one line at a time, and access to help. It's our preferred Python environment. Experts often use other editors, but unless you're one of them this is where you should start.

To start Spyder from the terminal, simply type `spyder` into the terminal and hit enter. We find it can sometimes be a little slow to start, but it should start up eventually. You should then see something that looks like this:


Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Tools View Help
C:\Users\dbackus\Dropbox\Documents\Python Scripts
Editor - C:\Users\dbackus\.spyder2-py3\temp.py
temp.py
Object inspector
Source Console
Object
Object inspector Variable explorer File explorer
IPython console
Console 1/A
Console History log IPython console
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 1 Column: 1 Memory: 19 %

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

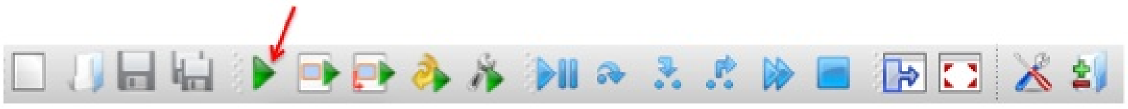
```
Python 3.5.0 |Anaconda 2.4.0 (64-bit)| (default, Oct 20 2015,
07:26:33) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

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.
%guiref   -> A brief reference about the graphical user
interface.

In [1]:
```

You see here that Spyder has a number of different components. It's overwhelming at first, but give it some time. The most important components are:

- **Editor.** This is on the left. We can write and edit programs here and save them to our hard drive.
- **Toolbar.** Above the editor, you'll see a row of buttons that we refer to as the toolbar. See the picture below. Among them are some green triangles. The big one (marked by the red arrow) runs the whole program -- whatever we have in the editor. The smaller ones run sections of code. More on this later.



- **IPython console.** This is on the right at the bottom -- look for the tab with this label. This is where output from our programs will show up. On startup it will display something like

  ```
  Python 3.5.0 |Anaconda 2.4.0 (64-bit)
  etc etc

  In [1]:
  ```

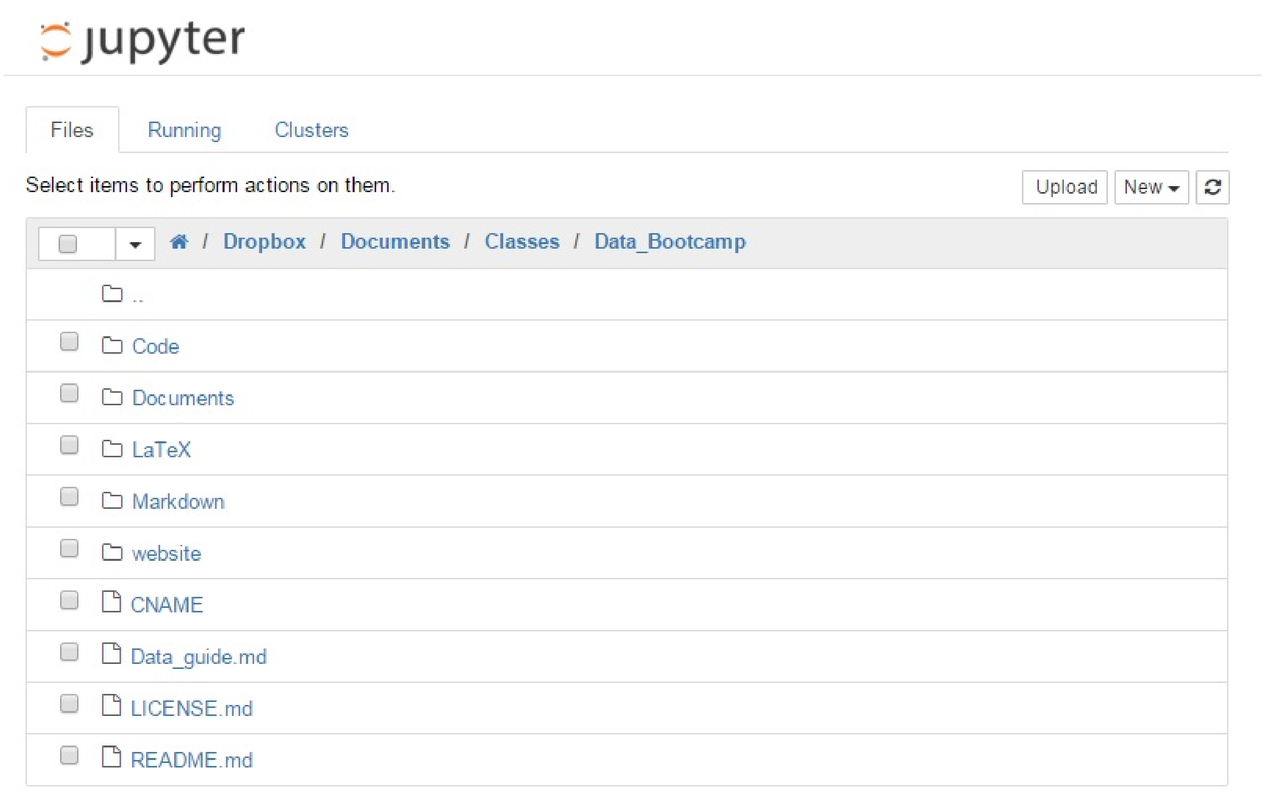
  Digression. Some Mac users report they have no IPython console. The solution: On the toolbar, go to View, then Panes, and make sure IPython console is checked. If that doesn't work, let us know.

- **Object inspector.** This is on the right at the top. We get Python documentation here, which is really useful.

We can move these windows around by dragging and dropping. If we mess up -- it happens to the best of us -- look for "View" at the top and click on "Reset window layout."

Jupyter. Jupyter is another graphical environment, which we use to create and run **Jupyter notebooks**. These notebooks combine code, output, words, and graphics. It's a convenient format for presenting our work to others and can be used as a project report. We'll use IPython notebooks in class in a few weeks. In the meantime, here are two examples.

To create or run a Jupyter notebook from the terminal, simply type `jupyter notebook` into the terminal and hit enter. It will open a tab in your default browser. (If you're not sure what that is, you'll soon find out.) In the browser tab, you'll see something like this:



Select items to perform actions on them.

/ Dropbox / Documents / Classes / Data_Bootcamp

..
Code
Documents
LaTeX
Markdown
website
CNAME
Data_guide.md
LICENSE.md
README.md

Exercise. Create a directory (folder) on your computer with the name `Data_Bootcamp` and store your programs there. If you're not sure how to do this, let us know. (And note well: There is an **underscore** `"_"` between "Data" and "Bootcamp", not a blank space.)

Run test programs

Let's run a test program -- the same one -- in Spyder and IPython/Jupyter and make sure everything works.

Spyder. Start up Spyder. (If you're not sure how to do that, go back to the previous section.) Once you have Spyder up and running:

- Create a Python code file. Click on "File" (upper left), then "New file." That should give you a new mostly empty file with some junk at the top that you can ignore or delete.
- Add code to file. Enter the following lines of code at the bottom of your file:

```
import sys      # import system module (don't ask)

print('\nWhat version of Python?\n', sys.version, '\n', sep='')

if float(sys.version_info[0]) < 3.0:
    raise Exception('Program halted, old version of Python. \n',
                    'Sorry, you need to install Anaconda again.')
else:
    print('Congratulations, you have Python 3!')
```

[If you're feeling lazy, you can make do with the first two lines on their own, but you won't get the messages we describe below.]

- Save your code. Click on File at the top left, then Save As, and save in the `Data_Bootcamp` directory under the name `bootcamp_test.py` (Python programs use the extension `.py`).
- Run it. Click on the green triangle above the editor and run your program.

The output appears in the IPython console in the lower right corner. If you get the message "Congratulations etc," you're all set. Pat yourself on the back and buy yourself a cold drink, you've earned it. If you get the message "Program halted, old version of Python, etc," you need to go back and install Anaconda again, this time **following the instructions exactly**! Yes, we know that's discouraging, but it's better to know that now than run into problems later. Have a cold drink anyway and catch your breath.

Comment (mtwn). We use the acronym **mtwn** to indicate material that is "more than we need," meaning it's safe to ignore. One of the challenges of learning to code is the overwhelming quantity of information. If you think you've exceeded your bandwidth, skip anything labeled mtwn.

More comments. All of these are mtwn, but we thought they would make the code we just entered less mysterious -- and give us a head start with Python programming. (i) Anything following a hash (#) is a comment and has no effect on what the program does. (ii) Blank lines are optional, but they make the code easier to read. (iii) The rest of the code checks the Python version (`sys.version_info`). If the version is less than 3.0, it prints an error message (`raise Exception`). Otherwise it prints the message "Congratulations, etc." (iv) The statements that begin with `raise` and `print` are indented exactly four spaces. That's a standard feature of Python. Anything else generates an error.

IPython. We prefer to write code in an editor and will stick with Spyder for a few weeks. Jupyter notebooks serve a different purpose: since they combine code with text and graphical output, they're well-suited for talks and reports. We can generally read through them more easily than naked code. Here are three more examples to make the point.

To run the same code in a Jupyter notebook, start the Jupyter notebook from the terminal. (If you're not sure what this means, go back to the previous section.) Once you have it up and running:

- Choose the directory. You should see the directory structure of your computer in Jupyter. Navigate to the `Data_Bootcamp` directory (folder) you created earlier.
- Create a Jupyter notebook. Click on the "New" dropdown menu in the upper right corner and choose Python 3. This will create a blank notebook and an empty cell, where you can enter words or code.
- Set the file name. To the right of the word Jupyter at the top, you'll see "Untitled". Change it to `bootcamp_test` .
- Enter code. Click on the dropdown menu below the word "Help" and choose Code. Then enter the code listed above in the empty cell.
- Run the code. Click on "Cell" at the top and choose Run All.

Output will appear in the same cell below your code. If it says "Congratulations etc." you're all set.

Let's go!

We're now ready to write and run Python programs in two environments. Take a bow.

Review

Exercise. We have seen both **code files** and **environments** for working with them. With this in mind, fill in the blanks in the table below and explain your answers to your neighbor.

Environment	File
MS Word	Word document
MS Excel	Excel file
Spyder	
	IPython notebook

Exercise. What version of Python are we using?

Exercise. Identify the editor, the IPython console, and the Object inspector in the Spyder picture above -- or your computer.

Resources

More on the Anaconda distribution and its contents:

- Anaconda download page and package list.
- Spyder documentation.
- IPython documentation. Look for the Pybonacci demo, it covers the basics in 5 minutes. You can also get help in Jupyter: click on "Help" at the top and choose "User Interface Tour."
- Links to other documentation and support. More than you'll ever want or need.

The data mentality

Overview. Thinking about data, ideas for projects. Things to remember: (1) Ideas aren't discovered, they're developed. (2) Ideas have friends: when you find one, there are others nearby.

Buzzwords. Questions, data, idea machines.

Code. Related examples

We study data because we want to learn something. But what? In our world, we might want to know:

- How is the US economy doing?
- What emerging market countries offer the best business opportunities?
- How do returns on US and European stocks compare?
- What college majors are paid the most?

You'll notice that **the starting point is a question**, something we'd like to know more about. We provide a toolkit for working effectively with data to find answers. Most of our examples are about economics and finance -- that's what we know -- but the same tools can be used to address any data we like.

Thinking about data

It's not that we have no lives or anything, but we think about data all the time. If we see an interesting graphic in *The Economist* -- or the *Wall Street Journal*, or the *New York Times* -- it triggers a series of questions.

- What did we learn from the graph?
- What else would we like to know?
- Where does the data come from?

Following up on these questions often leads to interesting insights. And it's fun.

Let's give it a try:

Exercise. The 538 blog has a nice summary of salaries of recent college graduates. Skip to the bottom to sort by major and play around. Answer these questions:

- What did you learn from their table?
- What else would you like to know?
- Where did they get their data?

Exercise. What kinds of things would you like to know more about? Think of this as improv, there are no bad answers.

Generating project ideas

One of our goals is for you to produce a piece of work -- data and graphics -- that you can show potential employers. There's nothing like a concrete example to show off your skill set. We still have lots of time, but it can't hurt to start thinking about it now.

Idea machines. How would we find a good project idea? That's not something you run across a lot in modern education, where our job is typically to absorb what's taught rather than come up with our ideas. So how would we get started?

We're looking for a topic that satisfies two conditions: (1) we find it interesting and (2) we have access to data related to it. We can start with either one, or with an existing example we would like to reproduce and extend:

- **Start with what interests you.** Economics, finance, marketing, emerging markets, movies, sports. You be the judge. Be specific: You want a topic, not a category.

- **Start with data.** Take a dataset you find interesting, ask what you might do with it. If you're not sure where to look, try our list of data sources.

- **Start with an example.** Find an analyst report, blog post, or graphic you like. Ask where the data comes from and think about whether you can replicate and/or extend it. The blogs listed on our data sources page are a good place to start.

If you're not sure how this works, watch Steve Levitt's video about working with company data. It's an entertaining and informative 50 minutes. Note specifically how he comes up with ideas for using the data he's given.

Keep in mind: we're not looking for a perfect idea. Perfection takes time, and we may never get there. Long experience has shown us:

- **Start small.** Small ideas often grow into bigger ones.

- **Ideas have friends.** If you have an idea, even a not very good one, it often triggers thoughts of other ideas, sometimes even better ones.

- **Ideas aren't discovered, they're developed.** Allow your ideas to mature, to evolve and improve. Like kimchi and red wine, they get better with time.

Common mistakes -- and how to fix them. We mean this in a good way, but in our experience there are a number of things students do that make this harder than it should be. Here's a list, with suggestions for overcoming them:

- **Reject an idea too soon,** before you've given it enough thought. Solution: Don't be critical too early, you don't want to inhibit your creativity. Collect ideas first, whittle them down later.
- **Choose a project that's too large**. Solution: Keep it simple. Think it over for a while, and choose a small part of a larger project that is interesting on its own. You can always do more later.
- **Your dataset doesn't have everything you want.** To be honest, that's pretty much every dataset we've ever seen. Solution: Make do with what you have.
- **Pick a dataset that's not available.** Solution: Start with what you have, ask what you can do with it. We call this the Jeopardy approach: start with the answer, come up with the question. If that fails, find another dataset.

Bottom line

Projects are less structured than most things you'll run across in school. It's challenging, at first, to work with so little structure, but most students find that the freedom to develop their own projects is one of the most rewarding things they can do.

Exercise. Write down three project ideas. Don't overthink this, one or two lines each will do.

Python fundamentals 1

Overview. Time to start programming! We work our way through some of the essentials of Python's core language. We do this in the Spyder coding environment. Part 1 of 2.

Python tools. Syntax, Spyder, calculations, assignments, strings, lists, built-in functions, objects, methods, tab completion, object inspector.

Buzzwords. Isn't that enough?

Trigger warning. Technical content, cannot be mastered without effort.

Code. Link

We're now ready to explore the rudiments of Python. We're going to **jump right in** to the deep end of the pool. For a couple weeks, you may feel like you've been dropped in a foreign country where you don't speak the language. You'll hear terms like "strings", "floats", "objects", "methods", and "tab completion". Some of these are words you know, but even so, they tend not to mean what you think they mean. (Insert Princess Bride joke.)

Don't panic, it's just jargon. If you put some effort into this over the next 2-4 weeks, you'll be fine. And ask questions. Really. **Ask lots of questions.**

The challenge and beauty of writing computer programs is that we need to be precise. If we mistype anything, the program won't work. Or it might seem to work, but the output won't be what we expect. In formal terms, the **syntax** -- the set of rules governing the language -- is less flexible than natural language (English, for example).

We mix Python concepts with an introduction to **Spyder**, the Python coding environment we described earlier. There we can not only run code, we can access help.

Reminders

Remind yourself about the following:

- Spyder. An environment for writing and running Python programs. Its components include an editor, an IPython console, and the Object explorer.

- `Data_Bootcamp` directory. The place in your computer where you store files for this course.

Exercise. Ask questions if you find any of these steps mysterious:

- Start Spyder.
- In Spyder, point to the editor, IPython console, and Object inspector.
- Open a new (empty) code file in Spyder and save it as `bootcamp_class_pyfun1.py` in your `Data_Bootcamp` directory/folder. This file will serve as your notes for this class.

The logic of Python programs

In a spreadsheet program such as Excel, we can connect cells to other cells. Then when we change one cell, any other cells connected to it update automatically.

Most computer programs, including Python programs, don't work that way. They run one line at a time, starting at the top of the program and working through the list of instructions until they reach the end or stop for some other reason. A program is just a detailed list of things we want the computer to do.

Most of the programs in this course have the structure:

- Input data.
- Manipulate the data until it's in the form we want.
- Produce some graphics that summarize the data in a compelling way.

Each of these bullet points is typically associated with a number of lines of code, possibly a large number, but that's the general idea.

Calculations in Spyder's IPython console

We'll do lots of numerical calculations. That's mostly what managing data is about: adding things up, dividing one thing by another, and so on. We'll do this initially in Spyder's **IPython console**, typically located in the lower right corner (look for a tab with this label).

To see how calculations work in Python, type these expressions in Spyder's IPython console **one at a time**:

```
2*3
2 * 3
2/3
2^3
2**3
log(3)
```

Type each one into the console, hit return, and look to see what happens. The first one multiplies 2 times 3, and (hopefully) gives us 6 as the answer. The input and output look like this in the console:

```
In [1]: 2*3
Out[1]: 6
```

The first line is our input, we typed it. The number in brackets `[1]` is a line number. We don't type it, it's there in the console to begin with. As we proceed the number [1] increases to [2], [3], and so on. The second line -- the one that starts `Out[1]` -- is the response or output Python produces.

The second calculation, `2 * 3`, does the same thing. The spaces around the * don't change the output. As a general rule, we can put spaces wherever we think they make the code more readable -- In fact, Python's style guide encourages you to use white space in a way that makes your code readable.

The third calculation is division. The input and output are

```
In [3]: 2/3
Out[3]: 0.6666666666666666
```

The fourth calculation, `2^3`, gives us

```
In [4]: 2^3
Out[4]: 1
```

Hmmmm. What just happened? We expected the answer to be 8 (2 to the power 3), but evidently it's not. The short answer is that the hat symbol `^` doesn't do exponents in Python, as it does in Excel. It does something else, which we won't go into.

That makes this is a good time to practice our **Google fu**:

Exercise. Use Google to search for "python exponents." Use what you find to compute 2 to the power 3. (Don't look below, that's cheating.)

We should find, after wading through the links, that exponents in Python are done this way:

```
In [5]: 2**3
Out[5]: 8
```

Exercise. What does the calculation `2 ** 3` produce?

Our last calculation is the log function. Entering `log(3)` generates the message: `NameError: name 'log' is not defined`. This is an example of a **syntax error**: we have used language that Python doesn't understand. Here the message is pretty clear: it doesn't know what `log` means. In other cases, the error message may be more mysterious. We can use functions like `log` and `sqrt` in Python, just as we do in Excel, but we need to import them specially. (And we will, but not yet.)

Exercise. What happens if you try to calculate the square root of 2 with `sqrt(2)`, as you would in Excel? How would you do it (You don't need to import it -- Hint: square root is a power)?

Assigning values to variables

Or maybe we should use scare quotes: "Assigning" "values" to "variables."

We'll start with examples and explain what they do. Type these two lines into the IPython console one at at time.

```
x = 2
y = 3
```

In each of these lines:

- The thing on the left is called a **variable**. In the first line, `x` is the variable. In the second, `y` is the variable.
- The thing on the right is a **value**. In the first line, `2` is the value. In the second, `3` is the value.
- The equals sign `=` **assigns** the value on the right to the variable on the left. Thus the first line assigns the value `2` to the variable `x`. The second assigns the value `3` to the variable `y`.

We call statements like these **assignments**: We assign a value to a variable.

We can see the results of these assignments by checking the contents of the variables `x` and `y`. In the IPython console, typing a variable and hitting return gives us its value. If we type `x` and `y`, one at a time, we get

```
In [7]: x
Out[7]: 2
In [8]: y
Out[8]: 3
```

So we see that the variables now contain the values we assigned them.

Variables are handy ways of storing values. We can use them in future calculations simply by using their names, just as we would use a cell address in Excel. Here's an example. Type this into the IPython console:

```
z = x/y
```

If we type `z` in the console and hit return, we get

```
In [9]: z
Out[9]: .666666666
```

What's going on here? We take `x` (which now has a value of 2) and divide it by `y` (which now has the value of 3) and assigns it to the variable `z`. The result is a computer's version of two-thirds.

Exercise. Type `w = 7` in the IPython console. What does the code `w = w + 2` do? Why is this not a violation of basic mathematics?

Exercise. Suppose we borrow 200 for one year at an interest rate of 5 percent. If we pay interest plus principal at the end of the year, what is our total payment? Compute this using the variables `principal = 200` and `i = 0.05`.

Exercise. Real GDP in the US (the total value of things produced) was 15.58 trillion in 2013 and 15.96 trillion in 2014. What was the growth rate? Express it as an annual percentage.

Exercise (challenging). Suppose we have two variables, `x` and `y`. How would you switch their values, so that `x` takes on `y`'s value and `y` takes on `x`'s?

Exercise (challenging). Type `x = 6` in the IPython console. We've reassigned `x` so that its value is now 6, not 2. If we type and submit `z`, we see

```
In [10]: z
Out[10]: .6666666666
```

But wait, if `z` is supposed to be `x/y`, and `x` now equals 6, then shouldn't `z` be 2? What do you think is going on?

Displaying results with the `print()` function

We saw that when we performed a calculation, such as `z = x/y`, we had to ask to see the result. The `print()` function gives us another way to do that. If we type `print(z)` in the IPython console, we get

```
In [11]: print(z)
0.6666666666666666
```

Evidently this displays the value of `z`, namely `0.6666666666666666`. We will use print statements a lot to track the progress of our code. We need to do this because Python -- and all other programming languages -- will not report back the results of what we ask it to compute unless we tell it to do so.

The print function displays whatever we include in parentheses after the word print: for example, `print(x)`. If we want to print more than one thing, we separate them with commas; for example, `print(x, y)`. That's the **general structure of functions** in Python: a function name (in this case `print`) followed by inputs ("arguments" or "parameters") in parentheses that are separated by commas. We usually refer to the `print()` function, with explicit parentheses, to remind ourselves that it requires input of some kind.

So if we want to verify the calculation of `z`, we can type `print(z)` in the IPython console. If we want to print all the calculations from the previous section, we can type `print(x, y, z)`:

```
In [12]: print(x, y, z)
2 3 0.6666666666666666
```

By default, the output is separated by spaces.

We'll use more complicated print statements than this, which we'll explain as we go. But if you see something you don't recognize, remember to **ask questions**.

Getting help in Spyder

If you want to know more about a function, there are two good ways to do it in Spyder. This works for any function, but using print as our examples we can:

- Type `print?` in the IPython console.
- Type `print` in the Object inspector.

The same approaches work for other functions. We use them both a lot. If they fail, either because there's no help or the help is incomprehensible, we fall back on Google fu.

Exercise. Run this statement: `print(x, y, z, sep='|')`. Use the output and Spyder's help to explain what the `sep` argument does.

Strings

We often work with non-numerical data, collections of characters that might include letters, numbers, or other symbols. Such things show up in a lot in data work, as variable names (GDP, income, volatility) and even as data (country or customer names, for example). We refer these as **strings**. This doesn't mean what we think it means. No, not the stuff we tie up packages with, but a "string" of characters like letters or numbers. It's one of many mysterious uses of ordinary words we'll run across as we learn to code. For more on this one, see here.

We create strings by putting characters between quotation marks: 'Chase', "Spencer", 'Sarah', "apple", and even '12' are all strings. Single and double quotes both work.

The last example is a confusing one, because it looks like a number. It's not. The number `12` is between quotes, so it's a string. If we try to use it as a number, it doesn't work. Try, for example, `'12'/3`. This generates the error: `TypeError: unsupported operand type(s) for /: 'str' and 'int'`. What this means is that we tried to divide a string (`'12'`) by an integer (`3`). That's no different to Python than trying to divide your name by three, it can't make sense of it.

We repeat: **a string is a collection of characters between quotes**. The characters can be pretty much anything. Therefore `12` is a number (no quotes), but `'12'` is a string.

Here are some other examples, which we assign to variable names for later use. Type them into Spyder's IPython console **one at a time**:

```
a = 'some'
b = 'thing'
c = a + b
d = '12.34'
```

What do you see? The first two are probably obvious: we assign the characters in single quotes on the right to the variables on the left.

The third line is something new: we add the string `some` to the string `thing`. What would you expect to get? Try `print(c)` to find out. That gives us the answer: `c = 'something'`. We've simply stitched the two strings together, one after the other.

Strings also allow us to produce better-looking output. In the previous section, for example, we can change the statement `print(z)` to `print('The value of z is ', z)`. The first argument (or input), `'The value of z is '`, is a string. The second argument, `z`, is a variable. Together they produce the output `The value of z is 0.6666666666666666`, which is clearer than the number `0.6666666666666666` on its own. Or we could spread this over two lines:

```
message = 'The value of z is '
print(message, z)
```

Here we've taken the components of the previous print statement and expressed them in two statements to make it more readable. (You might ask yourself: Which do you prefer? Why?)

Exercise. What is a string? How would you explain it to a friend?

Exercise. What happens if we run the statement: `'Chase'/2`? Why?

Exercise. This one's a little harder. Assign your first name as a string to the variable `firstname` and your last name to the variable `lastname`. Use them to construct a new variable equal to your first name, a space, then your last name. *Hint:* Think about how you would express a space as a string.

Exercise. Set `s = 'string'`. What is `s + s`? `2*s`? `s*2`? What is the logic here?

Running programs in Spyder

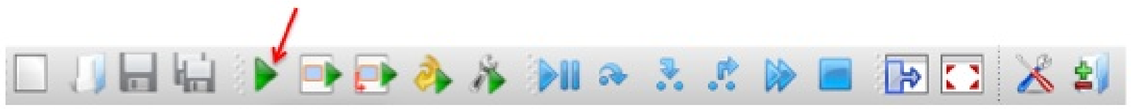
If we're writing longer programs, it's generally easier to type them into an editor where we can correct any mistakes we make, just as we do in a word processing program.

Let's give it a try. Type these commands into your file in the Spyder editor:

```
a = 'some'
b = 'thing'
c = a + b
print('c =', c)
```

Save the file (File, then Save).

Once we've saved the file, we can run it in Spyder by clicking on the large green arrow in the Spyder **toolbar** at the top of the editor window:



When we run the code, we see that the first three lines produce no output. The last one produces the output `c = something` in the IPython console.

Code cells in Spyder

Spyder has another cool feature we use a lot: we can carve out blocks of code ("cells") and run them separately. That way we can try out small pieces of code one at a time.

The idea is to put the separator `#%%` (hash, percent, percent) between blocks of code, called **cells**, so that we can run them separately. Consider the code:

```
a = 'some'
b = 'thing'
c = a + b
print('c =', c)
#%%
x = 2
y = 3
z = x/y
print('z =', z)
```

The separator `#%%` in the middle divides the file into two cells that we can run one at a time. That allows us to run and test blocks of code without running the whole program. It doesn't make much difference with code this simple, but in longer programs it can be a real time saver.

Here's how it works:

- In the Spyder editor, click on a code cell. The cell will indicate its selection with a darker background.
- Now go to the toolbar above the editor. The large green triangle runs the whole program. The one to its right displays the text "Run current cell" if you move the cursor to it. Click on it to run the selected cell.

Exercise. Copy or type the code above into your Python program. Save it. Run each cell, one at a time. Check the output to make sure it worked.

Add comments to your code

One of the rules of good code is that **we explain what we've done -- in the code**. In this class, we might think about writing code that one of our classmates can understand without help. These explanations are referred to as comments.

Add a comment with the hash character (#). Anything in a line after a hash is a comment, meaning it's ignored by Python. Here are some examples:

```
# everything that appears after this symbol (in the same line) is a comment!
# comments help PEOPLE understand the code, but PYTHON ignores them!
# we're going to add 4 and 5
4 + 5           # here we're doing it
print(4+5)      # here we're printing it
```

We often put comments like this in our code. Not quite this basic, but close. One of the unwritten "laws of nature" in programming is that code is read much more often than it is written 1 2 3. Writing informative comments will not only lead to others thanking you for saving them time, but you will find that you thank yourself very frequently.

Exercise moving forward. Practice writing comments **all the time**. Whenever you learn something new, write a comment explaining it in your code. It feels tedious, but the best coders always explain their work. It's a good habit to develop.

Single, double, and triple quotes

We typically define strings by putting characters between single quotes, as in `a = 'some'`. That will be our standard practice, but **Python treats single and double quotes the same**. We could have typed `a = "some"` (that is, with double quotes) with the same effect. The main reason for using single quotes is laziness: we don't have to hit the shift key. We're not ones to disparage laziness, but the point is that there's no difference between the two.

There are, however, some special cases where double quotes come in handy. On occasion, we use double quotes if the string includes a single quote:

```
f = "I don't believe it"
print(f)
```

The output of the print statement is `I don't believe it`, which includes a single quote as an apostrophe. This doesn't come up very often, in our experience, but there it is just in case.

Triple quotes are similar, but they can be used to define strings that go over several lines:

```
longstring = """
Four score and seven years ago
Our fathers brought forth. """
print(longstring)
```

This produces the output

```

Four score and seven years ago
Our fathers brought forth ...
```

The blank line comes from the empty space to the right of the first triple quote. And yes: we can make triple quotes from single quotes -- and this is more than we need.

We often put long quotes like this at the top of our programs. Officially they're strings, but unofficially they're comments. Here's an example from the start of our test program:

```
"""
Data Bootcamp test program checks to see that we're running Python 3.
Written by Dave Backus, March 2015
Created with Python 3.4
"""
```

Exercise. Put a comment like this at the top of your program.

Exercise. In the `Four score etc` code, replace the triple double quotes with triple single quotes. What happens?

Exercise. Fix this code:

```
bad_string = 'Sarah's code'
```

Exercise. Which of these are strings? Which are not?

```
apple
"orange"
'lemon84'
"1"
string
4
15.6
'32.5'
```

Lists

A Python list is what it sounds like: an ordered collection of items. The items can be lots of things: numbers, strings, variables, or even other lists.

Creating lists. We create lists by putting **square brackets around a collection of items** separated by commas. Here are some examples. Type each line of code into Spyder's IPython console and run them.

```
numberlist = [1, 5, -3]
stringlist = ['hi', 'hello', 'hey']
```

These are, of course, assignments: the lists on the right are assigned to the variables on the left.

We can also make lists of variables:

```
a = 'some'
b = 'thing'
c = a + b
variablelist = [a, b, c]
```

Or we can combine variables, numbers, and strings:

```
randomlist = [1, "hello", a]
```

Combining lists. We can combine lists (literally) by adding them. The statement `biglist = numberlist + stringlist` produces a list containing all the elements of `numberlist` plus all the elements of `stringlist`, giving us six items altogether. Type `print(biglist)` to make sure that's the case.

In contrast, the statement `biglist2 = [numberlist, stringlist]` produces a new list with two items: the lists `numberlist` and `stringlist`. It's what we might call a "list of lists." That's not something we're likely to do, to be honest. (We did it once, but that was an accident.) The point is simply that lists are flexible objects.

Exercise. How would you explain a list to a classmate?

Exercise. Add `print(numberlist)` and `print(variablelist)` to your code and note the format of the output. What do the square brackets tell us? The single quotes around some entries?

Exercise. Run the statements

```
mixedlist = [a, b, c, numberlist]
print(mixedlist)
```

What is the output? How would you explain it to a classmates?

Exercise. Suppose `x = [1, 2, 3]` is a list. What is `x + x` ? `2*x` ? Try them and see.

Tuples

Tuples are ordered collections of things in parentheses separated by commas. They're like lists but the syntax is different (parentheses rather than square brackets) and **they can't be changed** (experts would say they're "immutable"). We won't use them much but you're likely to run across them now and then in others' code.

Example. This is a tuple: `t = (1, 5, -3)` .

Python's built-in functions

We now have several kinds of **objects** to work with: numbers, strings, and lists. There are more on the way, but that's a good start. And yes, the formal term is really **objects**. But what can we do with them? Python has two basic ways to express things we do with objects: **functions** and **methods**. We'll talk about functions in this section and methods in the next one.

Python has a lot of basic "built-in" functions. We've already seen the `print()` function. Here are some others we've found useful.

The `len()` (length) function. This tells us the length of an object. We can work this one out for ourselves:

Exercise. Describe the output of typing each of these in the IPython console **one line at a time**:

```
len('hello')
len([1, 5, -3])
len((1, 5, -3))
len('1234')
len(1234)
len('12.34')
len(12.34)
```

What do you think is happening?

The `type()` function. One of our favs. This one tells us what kind of object we have. To see how it works, type the following into the IPython console one by one:

```
type(2)
type(2.5)
type('2.5')
type('something')
type([1, 5, -3])
type((1, 5, -3))
```

Think about this on your own for a minute. What do you think you'll get? How does it compare to the real output?

Not to kill the suspense, but here's what we should see:

- `type(2)` gives us the output `int`, which stands for "integer," a whole number like 1, 2, 3, and so on. Just to clarify: 2 is an integer. 2.5 is not.
- `type(2.5)` gives us `float`, a so-called "floating point number" like most of those we run across in Excel -- not a whole number.
- `type('2.5')` gives us `str`, a string because it's in quotes.
- `type('something')` also gives us `str`.
- `type([1, 5, -3])` gives us `list`.
- `type((1, 5, -3))` gives us `tuple`.

The type function is more helpful than you might guess. A lot of what we do in programming is deal with objects of different types and, when necessary, convert one type to another. The first step is to identify the type of the object of interest.

Exercise. Try each of these, one at a time, in the IPython console and explain the output:

```
type('a')
type('a1')
type('a1.0')
type(1)
type(1.0)
type('1')
type('1.0')
type([1, 5, 3])
type(['data', 'bootcamp'])
```

Exercise. Try `type(zoo)`. Why does it generate an error? What does the error mean?

Exercise. Set `zoo = ['lions', 'bears']` and try `type(zoo)` again. What do you get this time?

Changing types

If we have an object of one type, we can sometimes change it to another type. (Sometimes it doesn't make sense and Python tells us so.) The string `'12'` can be converted to the integer `12`, for example.

Converting strings to numbers. Suppose we have an object of one type (the string `'12.34'`) and want to use it as another (the number `12.34`). We can use the function `float()`, then use `type()` to check:

```
s = '12.34'
f = float(s)
type(f)
```

Here `s` is a string but `f` is the floating point number `12.34`.

The function `int()` lets us do the same for integers. Convert the string `'12'` to an integer, then check with `type()` again:

```
s = '12'
i = int(s)
type(i)
```

The result `i` is the integer 12.

Converting numbers to strings. Similarly, we can convert a number back to a string with `str()`:

```
s = str(12)
print('s has type', type(s))
t = str(12.34)
print('t has type', type(f))
```

Converting strings to lists. One more type conversion: We can convert a string to a list of its characters. For example, we convert the string `x = abc'` to the list `['a', 'b', 'c']` with `list(x)`. Run this code to see how it works:

```
x = 'abc'
y = list(x)
print(y)
```

Exercise. What happens if we apply the function `float` to the string `'some'`?

Exercise. How would you convert the integer `i = 1234` to the list `l = ['1', '2', '3', '4']` ?

Exercise. What is the result of `list(str(int(float('12.34'))))` ? Why? *Hint:* Start in the middle (the string `'12.34'`) and work your way out, one step at a time.

Exercise (challenging). This one is tricky, but it came up in some work we were doing. Suppose `year` is a string containing the year of a particular piece of data; for example, `year = '2015'` . How would we construct a string for the following year? *Hint:* Start by converting year to an integer.

Objects and methods

As we noted, lots of things in Python are **objects**. **Methods** are ready-to-go things we can do with these objects. The available methods depend on the object. A lot of Python is "object-oriented," which means we apply methods to objects to accomplish what you might think you need a function for. Trust us, the jargon is harder than just doing it.

Functions and methods differ primarily in their syntax:

- Syntax of a **function**: `function(object)`
- Syntax of a **method**: `object.method`

We used the former in the previous section and consider the latter here.

What methods are available to work with a given object? Take, for example, the list `numberlist = [1, 5, -3]` . To get the list of available methods, we use the IPython console and type:

```
numberlist.[tab]    # here you hit the tab key, don't type in the word "tab"
```

This wonderful piece of technology is referred to as **tab completion**. The ingredients here are the object (here `numberlist`), the period or dot, and the tab key. When you hit tab, a window will pop up with a list of methods in alphabetical order. In our example, the list starts like this:

```
numberlist.append
numberlist.clear
numberlist.copy
numberlist.count
```

If we want more information about a method, we can type `object.method?` in the IPython console or `object.method` in the object inspector. For the method `numberlist.append`, we get the description

```
Definition:  append(object)
Type:  Function of None module
L.append(object) -> None -- append object to end.
```

Well, that's pretty opaque, maybe we oversold this approach. What `append` does is add an item to the end of a list. Try using `append()`, then use `print()` to see the results:

```
numberlist.append(7)
print(numberlist)
```

That's another way to get information about a method: try it and see what happens.

Digression. We're trying to keep this simple, but we also want it to be accurate, so let us be more careful with the term **method**. Needless to say, this is **mtwn**. Tab completion gives us a list of two things: *attributes* (properties of the object) and *methods* (essentially functions with different syntax). The distinction isn't important to us, although we can tell a method because it comes with parentheses `()` (just like functions).

Example. Set `firstname = 'Chase'`. The method `lower` converts it to lower case. If we type `firstname.lower` into the object inspector, we see that it comes with parentheses for additional inputs. So we type `firstname.lower()` into the IPython console. The response is `'chase'`. The parentheses are there to provide additional inputs -- arguments, we call them. Without the parentheses, it doesn't work.

Exercise. Find a method to convert `firstname` to all upper case letters.

Exercise. This one also came up in our work. Suppose we have a variable `z = '12,345.6'`. What is its type? Convert it to a floating point number without the comma. Hint: Use tab completion to find a method to get rid of the comma.

Exercise. Run the code

```
firstname = 'John'
lastname  = 'Lennon'
firstlast = firstname + ' ' + lastname
```

Find a method to replace the n's in `firstlast` with asterisks.

Python 2 and 3

There's a lot of code around written in earlier versions of Python, most commonly Python 2.7. It's there because the people who wrote it started before Python 3 was up and running. Since we're starting from scratch, we are planting ourselves firmly in Python 3 territory. Still, you're likely to run across examples of Python 2 on the internet. The easiest way to tell the difference is the print command: `print(x)` in Python 3 was `print x` (no parentheses) in Python 2. There are some other differences (more technical than we care to discuss), which is why it's essential we all use Python 3.

Review

Work with your neighbor on these review exercises:

Exercise. Take turns with your neighbor explaining these terms: integer, float, string, list, tuple, function, method, tab completion.

Exercise. What types are these expressions? What lengths?

```
a = 1234
b = 12.34
c = '12.34'
d = [1, 2, 3, 4]
e = (1, 2, 3, 4)
```

Exercise. Describe the results of these operations:

```
a + b
a + c
d + d
d.append(a)
```

Exercise. Describe the results of these operations:

```
str(float('3.1416'))
str(int('3.1416'))
str(len('3.1416'))
str(len(3.1416))
```

Exercise. Set `name = 'Jones'`. Use (a) tab completion to find a method that coverts `name` to upper case (capital) letters and (b) the Object inspector to find out how to use that method. *Bonus:* How else can you get help in Spyder for methods and functions?

Exercise (challenging). Use tab completion and the Object inspector to find and apply a method to the string `name` that counts the number of appearances of the letter s. Use `name = 'Ulysses'` as a test case.

Exercise (challenging). Describe the result of `list(zip('1234','abcd'))`.

Resources

If you'd like another source for comparison, here are some good introductions to basic Python and related topics:

- Codecademy has an excellent Introduction to Python. You run Python in their online environment, which is really helpful when you're starting out. It uses Python 2, so the print statement has the form `print x` rather than `print(x)`. If we were to recommend one outside resource, this would be it. You should think seriously of working your way through it in parallel with this course. If you do, you can stop (as far as this course in concerned) when you get to Advanced Topics.
- Here's a list of free tutorials, but we think you can stop with the first one, Codecademy.
- The official Python tutorial is very good. It's also a good idea to get used to reading official documentation like this. There are times when it's unavoidable.
- Mark Lutz's Learning Python is a 1600-page monster that covers lots of things we won't use. But it's clear and thorough, and comes with the elusive Glenn Okun stamp of approval. He tells us the Kindle version comes with free updates.

These sources go well beyond what we do in this chapter, but we'll catch up with some of it later on.

Python fundamentals 2

Overview. More core Python. Part 2 of 2.

Python tools. Boolean variables, comparisons, conditionals (if, else), slicing, loops (for), function definitions.

Buzzwords. Code block, data structures, list comprehension, gotcha, PEP8.

Code. Link.

We continue our overview of Python's core language, which lays a foundation for the rest of the course. We go through the material quickly, since we're more interested in the general ideas than the details. You will feel like you're drinking from a fire hose, but it will sink in if you **stick with it**.

Reminders

Some things from previous chapters that we'll use a lot:

- Assignments and variables. We say we assign what's on the right to the thing on the left: `x = 17.4` assigns the number `17.4` to the variable `x`.
- Strings. Strings are collections of characters in quotes: `'this is a string'`.
- Lists. Lists are collections of things in square brackets: `[1, 'help', 3.14159]`.
- Number types: integers vs. floats. Examples of integers include -1, 2, 5, 42. They cannot involve fractions. Floats use decimal points: `12.34`. Thus `2` is an integer and `2.0` is a float.
- The `print()` function. Use `print(‘something’, x)` to display the value(s) of the object(s) in parentheses.
- The `type()` function. The command `type(x)` tells us what kind of object `x` is. Past examples include integers, floating point numbers, strings, and lists.
- Type conversions. Use `str()` to convert a float or integer to a string. Use `float()` or `int()` to convert a string into a float or integer. Use `list()` to convert a string to a list of its characters.

- Methods and objects. It's common in Python to work with objects using methods. We apply the method `justdoit` to the object `x` by typing `x.justdoit`.
- Spyder. An environment for writing Python programs. The various windows include an editor, an IPython console, and the Object explorer.
- Comments. Use the hash symbol `#` to add comments to your code and explain what you're doing.
- Tab completion. To find the list of methods available for a hypothetical object `x`, type `x.[tab]` in Spyder's IPython console -- or in an IPython notebook. We call that "tab completion."
- Help. We can get help for a function or method `foo` by typing `foo?` in the IPython console or `foo` in the Object explorer. Try each of them with the `type()` function to remind yourself how this works.

And while we're reviewing: Start Spyder, open a new file, and save as `bootcamp_class_pyfun2.py` in your `Data_Bootcamp` directory/folder.

Dictionaries

The term data structure refers to the organization of a collection of data. Strings and lists are examples. Here we look another one: dictionaries. We won't use them a lot, but when we do they're close to indispensible.

Dictionaries are (unordered) pairs of things defined by curly brackets `{}`, separated by commas, with the items in each pair separated by colon. For example, a list of first and last names:

```
names = {'Dave': 'Backus', 'Chase': 'Coleman', 'Spencer': 'Lyon', 'Glenn': 'Okun'}
```

If we try `type(names)`, the reply is `dict`, meaning dictionary. The components of each pair are referred to as the **key** (the first part) and the **value** (the second). In a real dictionary, the word is the key and the definition is the value. The keys must be unique, but the values need not be.

We access the value from the key with syntax of the form: `dict[key]`. In the example above, we get Glenn's last name by typing `names['Glenn']`. (Try it and see.)

We teach ourselves the rest:

Exercise. Print `names`. Does it come out in the same order we typed it?

Exercise. Construct a dictionary whose keys are the integers 1, 2, and 3 and whose values are the same numbers as words: one, two, three. How would you get the word associated with the key `2` ?

Exercise. Enter the code

```
d = {'Donald': 'Duck', 'Mickey': 'Mouse', 'Donald': 'Trump'}
print(d)
```

What do you see? Why do you think this happened?

Exercise. Describe -- and explain if possible -- the output of these statements:

- `list(names)` ?
- `names.keys()` ?
- `list(names.keys())` ?
- `names.values()` ?

Exercise. Consider the dictionary

```
data = {'Year': [1990, 2000, 2010], 'GDP':  [8.95, 12.56, 14.78]}
```

What are the keys here? The values? What do you think this dictionary represents?

Comparisons

Sometimes we want to do one thing if a condition is true, and another if it's false. For example, we might want to use observations for which the date is after January 1980, the country is India, or the population is greater than 5 million -- and not otherwise.

Python does this with **comparisons**, so called because they involve the comparison of one thing with another. For example, the date of an observation with the date January 1980. The result of a comparison is either `True` or `False` . If we assign a comparison to a variable, we refer to it as a **Boolean**, a name derived from the 18th century mathematician and logician George Boole. This gives us another type to add our collection: float, integer, string, list, dictionary, and now Boolean.

Let's try some simple examples to see what we're dealing with. Suppose we enter `1 > 0` in the IPython console. What does this mean? The input and output look like this:

```
In [1]:  1 > 0
Out[1]: True
```

The comparison `1 > 0` is interpreted as a question: Is 1 greater than 0? The answer is `True`. If we enter `1 < 0` instead,the answer is `False`.

A comparison is a Python object, but what kind of object is it? We can check with the `type()` function:

```
type(1>0)
```

The answer in this case is `bool` (that is, Boolean), the name we give to expressions that take the values `True` and `False`. (Actually, it says `<class 'bool'>`, but `bool` is enough to make the point.)

Python comes with a list of "operators" we can use in comparisons. You can find the complete set in the Python documentation, but common ones include:

- Equals: `==`
- Greater than: `>`
- Greater than or equals: `>=`
- Does not equal: `!=` (not equals).

We can reverse comparisons (change true to false and vice-versa) with the word `not`. For example:

```
In [2]: not 1>0
Out[2]: False
```

Think about that for a minute and evaluate it piece by piece. `1>0` returns `True` and `not True` is false. Additionally, remind yourself that spaces don't matter in Python expressions.

We can do the same thing with variables. Suppose we want to compare the values of variables `x` and `y`. Which one is bigger? To see how this works, we run the code

```
x = 2*3
y = 2**3
print('x greater than y is', x > y)
```

Here `x = 6` and `y = 8`, so the expression `x > y` (is `x` greater than `y`?) is false.

Exercise. What is `2 >= 1`? `2 >= 2`? `not 2 >= 1`? If you're not sure, try them in the IPython console and see what you get.

Exercise. What is `2 + 2 == 4`? How about `1 + 3 != 4`?

Exercise. What is `"Sarah" == 'Sarah'`? Can you explain why?

Exercise. What do these comparisons do? Are they true or false? Why?

```
type('Sarah') == str
type('Sarah') == int
len('Sarah') >= 3
```

Exercise (challenging). What do you think this code produces?

```
name1 = 'Chase'
name2 = 'Spencer'
check =  name1 > name2
print(check)
```

Run it and see if you're right. What type of variable is `check`? What is its value? Is Chase greater than Spencer?

Conditionals (`if` and `else`)

Now that we know how to tell whether a comparison is true or false, we can build that into our code. "Conditional" statements allow us to do different things depending on the result of a comparison or Boolean variable, which we refer to as a **condition**. The logic looks like this:

- if a condition is true, then do something.
- if a conditions is false, do something else (or do nothing).

To repeat: a condition here is a comparison or Boolean variable and is either true or false.

`if` **statements** tell the program what to do if the condition is true:

```
if 1 > 0:    # read this like "if 1>0 IS TRUE, then do the thing on the next line"
    print('1 is greater than 0')
```

The syntax here is precise:

- The `if` statement **ends with a colon**. That's standard Python syntax, we'll see it again. It's not optional.
- The code that follows is **indented exactly four spaces**. Also not optional. Spyder does it automatically.

Both of these features -- a colon at the end of the first line, indent the rest four spaces -- show up in lots of Python code. It's very compact, and the indentation makes the code easy to read.

Exercise. Change the code to

```
if 1 < 0:
    print('1 is less than 0')
```

What do you think happens? Try it and see.

Here's another example. Again, we do something if the condition is true, nothing if the condition is false. In this example, the condition is `x > 6`. If it's true, we print the number. If it's false, we do nothing. The code is

```
x = 7               # we can change this later and see what happens

if x > 6:
    print('x =', x)

print('Done!')
```

Here we've set `x = 7`, which makes the condition `x > 6` true. The `if` statement then directs the program to print `x`. The blank lines are optional; they make the code easier to read, which is generally a good thing. The statement `print('Done!')` is just there to tell us that the program finished.

Exercise. What happens if we set `x = 4` at the top? How do we know?

`else` statements tell the program what to do if the condition is false. If we want to do one thing if a condition is true and another if it is false, we would use `if` for the first and `else` for the second. The second part has been missing so far. Here's an example:

```
x = 7

condition = x > 6

if condition:
    print('if branch')              # do if true
    print(condition)
else:
    print('else branch')            # do if false
    print(condition)
```

The `else` statement adds the second branch to the decision tree: what to do if the condition is false. Try this with `x = 4` and `x = 7` to see both branches in action.

Exercise. Take the names `name1` and `name2`, both of them strings. Write a program using `if` and `else` that prints the name that comes first in alphabetical order. Test your program with `name1 = 'Dave'` and `name2 = 'Glenn'`.

Slicing strings and lists

We can access the elements of strings and lists by specifying the item number in square brackets. This operation is referred to as **slicing**, probably because we're slicing off pieces, like a cake. The only tricky part of this is remembering that **Python starts numbering at zero**.

Exercise. Take the string `a = 'some'`. What is `a[1]`?

What just happened? Python starts numbering at zero. If we want the first item/letter, we use `a[0]`. If we want the second, we use `a[1]`. And so on. We can summarize the numbering convention by writing the word `some` on a piece of paper. Below it, write the numbers, in order: 0, 1, 2, 3. Label this row "counting forward."

We can also count backward, but again Python has its own numbering convention. If we want the last letter, we use `a[-1]`. And if we want the one before the last one, we type `a[-2]`. In this case we get the same answer if we type `a[2]`. Both give us `'m'`.

Let's track this "backward" numbering system in our example. Below the "counting forward" numbers, start another row. Below the letter `e` write -1. As we move to the left, we type, -2, -3, -4. Label this row "counting backward."

Exercise. Take the string `firstname = 'Monty'` and write below it the forward and backward counting conventions. What integer would you put in square brackets to extract the third letter (`n`) under each system?

Exercise. Find the last letter of the string `lastname = 'Python'`. Find the second to last letter using both the forward and backward counting conventions.

We can do the same thing with lists, but the items here are the elements of a list rather than the characters in a string. The counting works the same way. Let's see if we can teach ourselves.

Exercise. Take the list `numberlist = [1, 5, -3]`. Use slicing to set a variable `first` equal to the first item. Set another variable `last` equal to the last item. Set a third variable named `middle` equal to the middle item.

More slicing

We've seen how to "slice" (extract) an item from a string or list. Here we'll show how to slice a range of items. For example, slice the last five characters from the string `c = 'something'`.

Recall that in Python we start counting at zero. If we want the first letter in `c`, we use `c[0]`. If we want the second, we use `c[1]`.

If we want more than a single letter, we need to specify both the start and the end. Let's try some examples and see what they do:

```
c = 'something'
print('c[1] is', c[1])
print('c[1:2] is', c[1:2])
print('c[1:3] is', c[1:3])
print('c[1:] is', c[1:])
```

Let's go through this line by line:

- The first print statement gives us `o`, the second letter of `something`. It's element 1 because we start numbering at zero.
- The next one does the same. Why not two letters? Let's try another one and see.
- The following line gives us `om`, the second and third letters. Why? Perhaps you figured it out. If not, this is the logic: the second number in `1:3`, namely `3`, is **one more than the end**. So the range `1:3` gives us the second and third letters. Confusing, for sure, but that's how it works.
- The last line has no second number. By convention it goes all the way to the end. The slice `c[1:]` goes from the second letter (the first number 1) to the end, giving us `omething`.

Some practice:

Exercise. Set `lastname = 'Python'`. Extract the string `'thon'`.

Exercise. Set `numlist = [1, 7, 4, 3]`. Extract the middle two items and assign them to the variable `middle`. Extract all but the first item and assign them to the variable `allbutfirst`. Extract all but the last item and assign them to the variable `allbutlast`.

Exercise. Take the string `c = 'something'`. What is `c[:3] + c[3:]`?

Loops over lists and strings (`for`)

There are lots of times we want to do the same thing many times, either on one object or on many similar objects. An example of the latter is to print out a list of names, one at a time. An example of the former is to find an answer to progressively higher degrees of accuracy.

We repeat an operation as many times as we need to get a desired degree of accuracy. Both situations come up a lot.

Here's an example in which we print all the items in a list, one at a time:

```
namelist = ['Chase', 'Dave', 'Sarah', 'Spencer']    # creates the list "namelist"
# below, the word "item" is arbitrary. End the line with a colon.
for item in namelist:    # goes through the items in the list one at a time
   print(item)    # indent this line exactly 4 spaces
# if there is code after this, we'd typically leave a blank line in-between
```

This produces the output

```
Chase
Dave
Sarah
Spencer
```

Note that `item` changes value as we go through the loop. It's a variable whose value actually varies.

We say here that we **iterate** over the items in the list and refer to the list as an **iterable**: that is, something we can iterate over. The terminology isn't important, but that's what it means if you run across it.

Exercise. What happens if we replace `item` with `banana` in the code above?

Example. We use a loop to compute the sum of the elements of a list of numbers:

```
numlist = [4, -2, 5]
total = 0
for num in numlist:
    total = total + num

print(total)
```

The answer (of course) is 7.

Exercise. Adapt the example to compute the average of the elements of `numlist`.

We can also run loops over the characters in a string. This one prints the letters in a word on separate lines:

```
word   = 'anything'
for letter in word:
    print(letter)
```

(You might think we could come up with a more interesting example than this. Sadly no, but we welcome suggestions.)

Example. Here's one that combines a `for` loop with an `if` statement to identify and print the vowels in a word:

```
vowels = 'aeiouy'
word   = 'anything'
for letter in word:
    if letter in vowels:
        print(letter)
```

(Adapted from SciPy lecture 1.2.) Describe what each line does as well as the overall result.

Example. What about the consonants? Note the word `not` below:

```
vowels = 'aeiouy'
word   = 'anything'
for letter in word:
    if letter not in vowels:
        print(letter)
```

Exercise. Take the list `stuff = ['cat', 3.7, 5, 'dog']`.

- Write a program that prints the elements of `stuff`.
- Write a program that tells us the `type` of each element of `stuff`.
- *Challenging.* Write a program that goes through the elements of `stuff` and prints only the elements that are strings; that is, the print the elements where function `type` returns the value `str`.

Loops over counters (`range()`)

We now know how loops work. Here's another version in which we loop over something a fixed number of times. For example, we might want to sum or average the values of a variable. Or value a bond with a fixed number of coupon payments. Or something.

The new ingredient is the `range()` function. `range(n)` gives us all the integers (whole numbers) from `0` to `n-1`. (If that sounds strange, remind yourself how slicing works.) And `range(n1, n2)` gives us all the whole numbers from `n1` to `n2-1`. We can use it in lots of ways, but loops are a prime example.

Some examples illustrate how this works:

Example. This is one of the simplest uses of `range()` in a loop:

```
for number in range(5):      # the variable "number" can be anything
    print(number)
```

It prints out the numbers 0, 1, 2, 3, and 4. (Ask yourself: Why doesn't it go to 5?) This is like our earlier loops, but `range(5)` has replaced a list or string as the "iterable."

Here's a minor variant:

```
for number in range(2, 5):
    print(number)
```

It prints out the numbers 2, 3, and 4.

Example. We compute and print the squares of integers up to ten. (Paul Ford comments: "Just the sort of practical, useful program that always appears in programming tutorials to address the needs of people who urgently require a list of squares.") We do that with a `for` loop and the `range()` function:

```
for number in range(1, 11):
    square = number**2
    print('Number and its square:', number, square)
```

Again we start at zero and work our way up to four.

Example. Here we compute the sum of integers from one to ten:

```
total = 0
for num in range(1, 11):
    total = total + num

print(total)
```

The answer is 55.

Example. Here's one that combines a loop and an `if` statement:

```
for num in range(10):
    if num > 5:
       print num
```

Exercise. Write a loop that computes the first five powers of two.

Example. Consider a bond that pays annual coupons for a given number of years (the maturity) and a principal of 100 at the end. The yield-to-maturity is the rate at which these payments are discounted. Given values for the coupon and the yield, the price of the bond is

```python
maturity = 10
coupon = 5
ytm = 0.05                 # yield to maturity

price = 0
for year in range(1, maturity+1):
    price = price + coupon/((1+ytm)**year)

price = price + 100/((1+ytm)**maturity)
print('The price of the bond is', price)
```

The answer is 100, which we might know because the coupon and yield are the same once we convert the latter to a percentage. Python gives us `99.99999999999997`, which is the computer's version of 100.

Digression. When we originally wrote this code, we used the variable name `yield` instead of `ytm`. Spyder marked this as `invalid syntax` with a warning sign to the left of the text. Evidently the name `yield` is reserved for something else. As general rule, it's a good idea to pay attention to the hints like this.

Loop with condition. Sometimes we want to go through a loop until some condition is met. This combination of a loop and a condition requires an extra level of indenting. It also introduces a new ingredient: the `break` statement, which tells Python to exit the loop.

Example. Suppose we want to compute the sum of integers until the sum reaches 100. We could use the code

```python
maxnum = 20             # guess of number above our limit

total = 0
for num in range(maxnum):
    total = total + num
    if total > 100:
        break              # exit loop

print('At num =', num, 'we had total =', total)
```

The `if` statement starts with a colon and the statement following it (`break`) is indented four spaces more (eight in total). `break` is a special command that ends a loop early.

Let's review:

Exercise. In Portugal and Greece, policymakers have suggested reducing their debt by cutting the coupon payments and extending the maturity. How much do we reduce the value of the debt if we reduce the coupons to 2 and increase the maturity to 20?

Exercise. Consider the list `namelist = ['Chase', 'Dave', 'Sarah', 'Spencer']`. Write a loop that goes through the list until it reaches a name than begins with the letter `S`. At that point it prints the names and exits the loop.

List comprehensions

That's a mouthful of jargon, but the idea is that we can create lists (and do related things) using implicit loops that we refer to as **list comprehensions**. This is incredibly useful and shows up a lot in Python code.

Example. Consider the loop above that prints out the elements of the list `namelist` one at a time:

```
namelist = ['Chase', 'Dave', 'Sarah', 'Spencer']
for item in namelist:
    print(item)
```

A list comprehension gives us more compact syntax for the same thing:

```
[print(item) for item in namelist]
```

As with loops, the variable `item` is a dummy: we can use any name we wish. Replace `item` with your pet's name to see for yourself.

Example. Take the list `fruit = ['apple', 'banana', 'clementine']`. Here's a list comprehension that creates a new list of capitalized fruits:

```
FRUIT = [item.upper() for item in fruit]
```

Try it and see. And think about the loop version a minute to see what we've avoided.

Example. We can do the same with a condition. This one takes the list `fruit` and creates a new list that contains only those names with six letters or less:

```
fruit6 = [item for item in fruit if len(item)<=6]
```

Exercise. Take the list `fruit` and create a new list with the first letter capitalized. *Hint:* What method would you use to capitalize a string?

Exercise. Take the list of growth rates `g = [0.02, 0.07, 0.07]`. Write a list comprehension that multiplies each element by 100 to turn it into a percentage.

Defining our own functions

It's easy to create our own functions -- experienced programmers do it all the time. A common view is that we should never copy lines of code. If we're copying, we're repeating ourselves. What we should do instead is **write a function once and use it twice**. More than that, breaking a long program into a small number of functions makes the code easier for others to read, which is always a good thing. As we become more comfortable with Python we'll use functions more and more.

Defining functions. The simplest functions have two components: a **name** (what we call it) and a list of **input arguments**. Here's an example:

```
def hello(firstname):                   # define the function
    print('Hello,', firstname)

hello('Chase')                          # use the function
```

Let's go through this line by line:

- The initial `def` statement defines the function, names it `hello`, identifies the input as `firstname`, and ends with a colon (:).
- The following statement(s) are indented the usual four spaces and specify what the function does. In this case, it prints `Hello,` followed by whatever `firstname` happens to be. Python understands that the function ends when the indentation ends.
- The last line "calls" the function with input `Chase`. Note that the name in the function's definition and its use need not be the same.

Function returns. Our function `hello` has a name (`hello`) and an input argument (`firstname`), but returns no output. Output would create a new value that Python could call later in the code, like when you set `x = 2` then used `x` later on. Here we print something but produce no other output.

In other cases, we might want to send output back to the main program. We do that with a **return** statement, a third component of a function definition. Here's an example

```
def squareme(number):
    """
    Takes numerical input and returns its square
    """
    return number**2        # this is what the function sends back

square = squareme(7)        # assign the "return" to variable on left
print('The square is', square)
```

And here's another one:

```
def combine(first, last):
    """
    Takes strings 'first' and 'last' and returns new string 'last, first'
    """
    lastfirst = last + ', ' + first
    return lastfirst                    # this is what the function sends back

both = combine('Chase', 'Coleman')      # assign the "return" to both
print(both)
```

Here we return the string `'Coleman, Chase'` and assign it to the variable `both`. Note, too, the comment in triple quotes at the top of the function. That's standard procedure, we recommend it.

The return is an essential component of many functions. Typically when we read the documentation for a function or method, one of the first things we look for is what it returns.

Exercise. Create and test a function that returns an arbitrary power of 2: the input `n` (an integer) returns the output `2**n`. Use `n=2` and `n=5` as test cases.

Exercise. Create and test a function `nextyear` that takes an integer year (say 2015) and returns the following year (2016).

Exercise. Use the Object inspector to get the documentation for the built-in function `max`. If the input is a list of two or more numbers, what does `max()` return?

Exercise (challenging). Create and test a function that takes a string year (say, `'2015'`) and returns a string of the next year (say, `'2016'`).

Programming style

Yes, style counts. We're not only trying to get something done, we're also communicating with others who may look at our code and possibly use it. A clear style makes that communication more effective.

With that in mind, here are some guidelines we've found useful:

- Put an overall summary of your program at the top in triple quotes. This should include both the purpose of the program and your name. Your email address is optional.
- Lines should be no longer than 79 characters.
- Skip two lines before and after a function definition.
- Skip lines here and there where you think it makes sense.
- Use comments whenever something isn't immediately obvious (and sometimes even when it is).

You can find more along these lines in the classic "PEP8" and Google's style guide.

Some programmers are religious about this. We'd say simply that we want to make our code readable by others.

There's one other thing we often do. If we find documentation online -- at Stack Overflow, for example -- we put a link to it in the code for future reference.

Review

Exercise. What type is each of these expressions? What length?

- `'abcd'`
- `[1, 3, 5, 7]`
- `{1: 'one', 2: 'two'}`
- `123`
- `123.0`
- `list('abcd')`
- `range(3)`
- `list(range(3))`

Exercise. Which of the following are `True` and which are `False` ?

- `2 >= 1`
- `2 >= 2`
- `2 != 2`
- `'this' == "this"`
- `'Chase' < 'Dave'`
- `'Chase' < 'Dave' or 'Spencer' < 'Glenn'`
- `'Chase' < 'Dave' and 'Spencer' < 'Glenn'`

Exercise. Take the object `numbers = {1: 'one', 2: 'two'}` . What type is it? Extract the keys as a list. Extract the values as a list.

Exercise. Write a program that prints the last letter of each item in the list `names = ['Chase', 'Dave', 'Sarah', 'Spencer']`. **Bonus (optional):** Print the last letter only if it's a vowel.

Exercise. Write a function `lastletter` that extracts the last letter from a string. Use `'Gianluca'` as your test case.

Exercise (challenging). Take the list of bond yields `y = [0.01, 0.02, 0.03]` for maturities of one, two, and three years.

- What happens if you try to multiply all of them by 100 with `100*y`?
- How would you accomplish the same task (multiply all the elements of `y` by 100) with a loop?
- How would you accomplish the same task (multiply all the elements of `y` by 100) with a list comprehension?

Exercise (challenging). Start with the lists `l1 = [1, 2, 3]` and `l2 = ['one', 'two', 'three']`.

- What does `list(zip(l1,l2))` do?
- What does `dict(list(zip(l1,l2)))` do?
- Create a list that contains only the number names in `l2` that have three letters.
- Write a list comprehesion that constructs the list of tuples `[(1, 1), (2, 4), (3, 9)]`.
- Convert the list of tuples into a dictionary.

Resources

See the resources in the previous chapter, especially Codecademy. If you work your way up to Advanced Topics, you'll be in good shape for anything that follows.

Additional resources:

- The official Python Tutorial has a nice introduction to "control flow language" that includes comparisons, conditional statements, and loops.
- CodingBat has a great collection of exercises. Significantly more demanding than ours. Runs online.
- Udacity has a free Introduction to Computer Science course that covers Python from a more technical perspective. Recommended for people who want to understand the structure and logic of the language.
- This is way more about comprehensions than you ever wanted to know, but it's so beautifully done you might want to take a look.

One last one, but only if you're curious about floating point numbers. Ok, that's approximately no one. Try this anyway and think about what's going on (it also gives us an idea that we don't want to check for strict equality for floating point numbers -- Better to check whether they are close enough):

```
0.1 + 0.2 == 0.3
```

False? More here.

Python packages

Overview. We introduce "packages" -- collections of tools that extend Python's capabilities. We describe tools used to update and install Python packages.

Python tools. packages, Conda, Pip.

Buzzwords. Command line.

Applications. Seaborn, Bokeh, pandas-datareader, plotly

Python is not just a programming language, it's an open source collection of tools that includes both core Python and a large collection of packages written by different people. The word **package** here refers to plug-ins or extensions that expand Python's capabilities. Terminology varies. What we call a package others sometimes call a "library." The term "module" typically refers to a subset of core Python or one of its packages. You won't go far wrong to use the terms interchangeably.

The standard Python packages are well written, well documented, and well supported. They have armies of users who spot and correct problems. Some of the others less so. For the most part, we stick to the standard packages, specifically those that come with the Anaconda distribution.

Some of the leading packages for numerical ("scientific") computation are

- **Pandas.** The leading package for managing data
- **Matplotlib.** The leading graphics package. We'll use it extensively.
- **NumPy.** Tools for numerical computing. In Excel the basic unit is a cell, a single number. In NumPy the basic unit is a vector (a column) or matrix (a table or worksheet), which allows us to do things with an entire column or table in one line. This facility carries over to Pandas.

All of these packages come with the Anaconda distribution, which means we already have them installed and ready to use.

Pandas is an essential part of data work in Python. Its authors describe it as "an open source library for high-performance, easy-to-use data structures and data analysis tools in Python." That's a mouthful. Suffice it to say that we can do pretty much everything in Pandas that we can do in Excel -- and more. We can compute sums of rows and columns, generate new rows or columns, construct pivot tables, and lots of other things. And we can do all this with much larger files than Excel can handle.

More than you need

Here are some other packages you might run across:

- **Seaborn.** A "wrapper" (isn't that a great term?) for Matplotlib that makes it easier to use.
- **Statsmodels.** The basic statistics package -- This allows us to do things like regression.
- **Scikit-learn.** A package for "machine learning," which is the name computer scientists give to data work. CS people have done some cool things with data. They're also really good at naming things: machine learning, visualization, support vector machines, random forests.
- **NLTK.** The so-called Natural Language Toolkit processes text. Here "natural language" means "words." Investors, for example, might process the words used in financial reports to detect mood or sentiment. One of our students is using NLTK to analyze tweets. The big-picture idea is that data can be anything, not just numbers.
- **Beautiful Soup** and **Scrapy.** Extract ("scrape") data from web sites.
- **Django.** A popular tool for website development.

We won't use most of them them, but they're in Anaconda too. Feel free to give them a try. If you do, please report back on your experience.

Importing packages

In Python we need to tell our program which packages we plan to use. We do that with an `import` statement.

Here are some examples applied to a mythical package `xyz` and mythical function `foo`:

- `import xyz`. This imports the package `xyz`. A function `foo` in package `xyz` is then executed with `xyz.foo()`
- `import xyz as x`. This also imports the package `xyz`, but under the abbreviation `x`. Here `foo` is executed with `x.foo()`. This is the most common syntax and the one we'll generally use. With Pandas, for example, the standard import statement is `import pandas as pd`.

- `from xyz import foo`. This imports the `foo` function directly so it can be called with `foo()`. Note in this case that if we had another function `bar` in the `xyz` package we wouldn't not be able to call it without importing the `bar` function like we did `foo`, or imported the `xyz` module itself using one of the methods above.
- `from xyz import *`. This imports all the functions and methods from the package `xyz`, but here the function `foo` is executed simply by typing `foo`. We don't usually do this, because it opens up the possibility that the same function exists in more than one package, which is virtually guaranteed to create confusion.

We'll see these examples repeatedly:

```
import pandas as pd                    # data package
import matplotlib.pyplot as plt        # graphics package
```

You might also go through earlier chapters and identify the `import` statements you find. By convention, they are placed at the top of the program. What packages or modules have we used? What do they do?

Some fine points:

- Redundancy. What happens if we issue an import statement twice? Answer: Nothing, no harm done.
- Jokes. These are programmer jokes, which some might see as a contradiction, but try them and see what happens:

  ```
  import this
  ```

  ```
  import antigravity
  ```

- Versions. We can check the version number of a package with `package.__version__`. To check the version of Pandas you have installed, try

  ```
  import pandas as pd
  print('Pandas version ', pd.__version__)       # these are double underscores
  ```

  This can be helpful if we're trying to track down an error.

Exercise. Import Pandas. What version do you have?

Exercise. What happens if we import Pandas twice under different names, once with `import pandas as pd` and once with `import pandas as pa`? Write a short program that tests your conjecture. *Hint:* Use what have we done with Pandas so far.

Updating Python: Conda and Pip

Warning: This is a little terse, but we think it does the trick.

Python has some handy tools for installing and updating the core language and its packages. We start with **conda**, the tool used to update the components of the Anaconda distribution, then move on to the traditional Python tool, **pip** (acronym for "pip installs packages").

As usual, the idea is to get things started. We provide links to more extensive documentation at the end.

These tools run on the **command line**, the old-school approach to computing that predates the graphical interfaces we use today (Mac OS, Windows, etc). Picture the computer screens in "War Games." Many of the graphical interfaces used in the computing world run things on the command line behind the scenes, so it's often better to go straight to the source and run things on the command line ourselves.

We have been doing this to open spyder, so we should have an idea of how to do this.

Conda

Conda is the Anaconda tool. We use the Anaconda distribution of Python, which we think is the most user-friendly version out there. It comes with a variety of packages installed, including our favorites: Pandas and Matplotlib.

Basics. Conda has a number of tools for managing our Python installation. Try each of these commands (from the command line -- *not* inside Ipython or spyder) and see what happens:

```
conda info
conda help
```

The first one tells us the version of Conda we have. It knows whether we installed the Windows, Mac, or Linux versions, and tells us where Anaconda is installed on our computer.

The second line gives us a list of Conda commands. We'll focus on `update` and `install`, but first let's see what we've got installed.

Exercise. Enter `conda list`. How many packages do you have installed? Verify that Pandas and Matplotlib are among them. What versions do you have?

Exercise. Go to the Anaconda package list and verify that it contains Seaborn. Use `conda list` to see if we already installed it. (Probably not, but it's worth checking.)

Updating. The next step is to update our Python installation. Updating with Conda is simpler than installing Anaconda again, and less likely to lead to trouble. First we update Conda:

```
conda update conda
```

This checks to see how our installation compares to the current version. If we're up to date, it will respond: `All requested packages already installed.` If not, we get a long list of packages and the question: `Proceed ([y]/n)?` (Or just hit return and yes will be assumed.)

Next we update Anaconda:

```
conda update anaconda
```

If we're up to date, we get the same response as before. If not, it will ask us to type `y` (yes) or `n` (no) to update our version. Type `y` to update. This can take a while, you might want to get a cold drink while you're waiting.

Exercise. Update Conda and Anaconda on your computer.

Installing new packages. Next up: installing a new packages. If we want to install a package, and it's part of the Anaconda distribution, we install it with

```
conda install [package]
```

Here `[package]` stands for whatever package we have in mind.

Exercise. Install the graphics package Seaborn with

```
conda install seaborn
```

Exercise. Install the data input package `pandas-datareader`. This will be needed in the next chapter, so do not skip this exercise!

Pip

Pip is the traditional Python package management tool. While Conda accesses packages included in Anaconda, Pip accesses a larger number of packages stored in the **Python Package Index**, commonly referred to as PyPI or the Cheese Shop.

Pip works pretty much the same way Conda works. We type things like this at the command line prompt:

```
pip help
pip list
pip install [package]
```

(This should remind you of what we just did with `conda`?) We won't speak more about it, but if you use a package that's not in Anaconda, this is the fallback.

Exercise. Use pip to install the `plotly` package. It is another package for creating plots in Python and we will use it later in the course.

Resources

The documentation has way more than we want or need, but the Conda cheatsheet isn't bad.

Pandas 1: Introduction

Overview. We introduce the Python package pandas and its core type the DataFrame. pandas is the Python package devoted to data management. In this chapter, we cover how to create a DataFrame and discuss some of its properties and methods.

Python tools. pandas, DataFrame, Series

Buzzwords. DataFrame, Series

Applications. US GDP

Code. Link.

The goal of this lecture is to introduce you to some of the basic concepts needed to understand how to do data analysis in Python. In particular, we introduce you to the pandas package and discuss its two core types DataFrames and Series.

It is worth noting that the pandas package could be considered more **high-level** than core Python in the sense of taking a lot of the programming details out of our hands. That makes it easier to use -- lots of things are automated -- but in some cases also a bit more mysterious. All together, though, it's an incredibly powerful collection of data tools which will make your life throughout this class (and beyond) much easier.

Reminders

- Objects and methods. Recall that we apply the method `justdoit` to the object `x` with `x.justdoit()` .
- Help. We get help in Spyder from both the IPython console and the Object inspector. For the hypothetical `x.justdoit` , we would type `x.justdoit?` in the IPython console or `x.justdoit` in the Object inspector.
- Data structures. That's the term we use for specific organizations of data. Examples are lists, tuples, and dictionaries. Each has a specific structure and a set of methods we can apply. List are (ordered) collections of objects between square brackets: `numberlist =`

`[1, -5, 2]`. Dictionaries are (unordered) pairs of items between curly brackets: `namedict = {'Dave': 'Backus', 'Chase': 'Coleman'}`. The first item in each pair is the "key," the second is the "value.""

- Integers, floats, and strings. Three common types of data.
- Function returns. We refer to the output of a function as its **return**. We would say, for example, that the function `type(x)` `return`s the type of the input object `x`. We capture the return with an assignment: `xtype = type(x)`.
- Packages. Packages are collections of tools -- functions and types/methods -- that extend Python's capabilities. We import a package using an `import` (e.g. `import pandas`) statement or some combination of `import`, `from`, and `as` (e.g. `import pandas as pd` or `from pandas import DataFrame`).

First Look at DataFrames

The entire pandas package is oriented around the idea of a DataFrame, so it is natural to begin our description of the package there. A DataFrame is similar to a sheet of data in excel (or to an R `data.frame` if you have programmed in R before). Let's create one so that we can see what it looks like (don't forget to run `import pandas as pd` first -- all of our examples will be based on you having previously done this). Let's create our first DataFrame by using data from a dictionary.

```
# This dictionary is similar to one that we saw earlier in the class
# It represents GDP/CPI data at 10 year intervals from 1990 to 2010
df = pd.DataFrame({"GDP": [5974.7, 10031.0, 14681.1],
                   "CPI": [127.5, 169.3, 217.488],
                   "Year": [1990, 2000, 2010],
                   "Country": ["US", "US", "US"]})
print(df)
```

What do you see on your screen? The print command on our DataFrame should have displayed something that looks like this:

```
       CPI Country      GDP  Year
0  127.500      US   5974.7  1990
1  169.300      US  10031.0  2000
2  217.488      US  14681.1  2010
```

We can verify that this type is indeed a DataFrame (and that we haven't been lying to you about what you created) by simply asking Python to tell us its type:

```
type(df)
```

returns

```
pandas.core.frame.DataFrame
```

The text prior to DataFrame just tells us some things about where this type lives within pandas -- You can safely ignore it. Now we should talk about what makes up a DataFrame.

- At the top of each of the columns of the dataframe, you should see "CPI", "Country", "GDP", and "Year" (which were the keys to our dictionary). These are known as the *column labels* or *column names*.
- To the left of the columns, we see a 0, 1, and 2. These numbers are elements of the *index* (or *row labels*).
- Finally, inside the *index* and *column labels*, you should see some data (which corresponds to the values of our dictionary). These are known as the *values*. We refer to the data going down a column as a *column* and the data going across a row as a *row*.

Typically columns are variables and the column labels give us their names. In our example, the second column has the name `Year` and its values follow below it. The rows are then observations, and the row labels give us their names. This is a standard setup and we'll do our best to conform to it. If the data come in some other form, we'll try to convert it.

We can get all of the relevant information from our DataFrame by accessing the following properties:

Dimensions. We access a DataFrame's dimensions -- the numbers of rows and columns -- using `df.shape` . Here the answer is `(3, 4)` , so we have 3 rows (observations) and 4 columns (variables).

Columns and indexes. We access the column and row labels directly. For the DataFrame `df` we read in earlier, we extract column labels with the `columns` method: `df.columns` . That gives us the verbose output `Index(['CPI', 'Country', 'GDP', 'Year'], dtype='object')` . If we prefer to have them as a list, we can use a `tolist` method `df.columns.tolist()` . That gives us the column names as a list: `['CPI', 'Country', 'GDP', 'Year']` .

The row labels are referred to as the **index**. We extract them by accessing `df.index` . That gives us the verbose output `RangeIndex(start=0, stop=3, step=1)` . We can convert it to a list by using the same `tolist` method, `df.index.tolist()` , which gives us `[0, 1, 2]` . In this case, the index is not part of the original data; Pandas inserted a counter for us. As usual in Python, the counter starts at zero.

Column data types. Pandas allows every column (typically a variable) to have a different data type, but the type must be the same within a column. With our DataFrame `df`, we get the types by using `df.dtypes`:

```
CPI          float64
Country       object
GDP          float64
Year           int64
dtype: object
```

Evidently `Year` is an integer and both `CPI` and `GDP` are floats. They're no different from the types of numbers we came across in the previous chapter. The column, `Country`, is different though. pandas labeled this one as having type `object`. Object is the name Pandas gives to things it can't turn into numbers -- in our case, strings. Sometimes, as here, that makes sense: country names like `US` are naturally strings. But in many cases we've run across, numbers are given the dtype object because there was something in the data that didn't look like a number. We'll see more of that later on.

Transpose columns and rows. If we want to rotate the DataFrame, exchanging columns and rows, we use the `transpose` method: `df.transpose()` or (more succinctly) `df.T`. Let's do that with the DataFrame `df` we read in earlier:

```
dft = df.T
print('\n', dft)
```

The result is

```
CPI        127.5   169.3  217.488
Country       US      US       US
GDP       5974.7   10031  14681.1
Year        1990    2000     2010
```

It doesn't make much sense in this case, but in others we'll find it helpful.

Exercise. Let's apply what we have learned to the DataFrame generated by this code:

```
data = {'countrycode': ['CHN', 'CHN', 'CHN', 'FRA', 'FRA', 'FRA'],
        'pop': [1124.8, 1246.8, 1318.2, 58.2, 60.8, 64.7],
        'rgdpe': [2.611, 4.951, 11.106, 1.294, 1.753, 2.032],
        'year': [1990, 2000, 2010, 1990, 2000, 2010]}
pwt = pd.DataFrame(data)
```

- What kind of object is `data`?

- What are the dimensions of `pwt` ?
- What dtypes are the variables? What do they mean?
- What does `pwt.columns.tolist()` do? How does it compare to `list(pwt)` ?
- *Challenging.* What is `list(pwt)[0]` ? Why? What type is it?
- *Challenging.* What would you say is the natural index? How would you set it?

Operating on DataFrames

So we have a DataFrame `df` whose columns are variables. One of the great things about pandas is that we can do things with every observation of a variable in one statement.

Variables = Series. In the DataFrame `df` we created earlier, if we wanted to refer to only one of the columns (variables), for example `GDP` , we write `df["GDP"]` .

```
print(df["GDP"])
```

If we ask what type this is, with

```
print(type(df['GDP']))
```

We find that it's a `pandas.core.series.Series` -- Similar to as with DataFrames, we refer to this as **series** for short. A series is essentially a DataFrame with a single variable or column. Anytime we ask for a single variable back, pandas will return us a Series, but if we ask for multiple rows (`df[["GDP", "CPI"]]`) then pandas will return us a DataFrame.

Series are useful because we can do addition, subtraction, division, and multiplication with them. Here are some examples

```
print(df["GDP"] + df["GDP"])
print(df["GDP"] - df["GDP"])
print(df["GDP"] / df["CPI"])
print(df["CPI"] * df["CPI"])
```

How do you think these operations are happening? How does Python know which two elements to add together etc?

Additionally, we can do operations on a Series with integers or floats. For example

```
print(df["GDP"] / 10000)
```

Construct new variables from old ones. Now that we know how to refer to a variable and about some of the operations we can do between variables, we can construct others from them. We construct two as an example

```
df['RGDP'] = df['GDP']/df['CPI']
df['GDP_div_1000'] = df['GDP'] / 1000
```

The first line computes the new variable `RGDP` as the ratio of `GDP` to `CPI` (here it does division element by element). The second computes `GDP_div_1000` as `GDP` divided by `1000` (here it divides everything by 1000).

These statements do two things: they perform the calculation on the right, and they assign it to the variable on the left. The second step adds the new variables to the DataFrame. The statement `print('\n', df)` now gives us

```
       CPI  Country      GDP   Year        RGDP   GDP_div_1000
0  127.500       US   5974.7   1990   46.860392         5.9747
1  169.300       US  10031.0   2000   59.249852        10.0310
2  217.488       US  14681.1   2010   67.503035        14.6811
```

If we step back for a minute, we might compare this to Excel. If `GDP` and `CPI` are columns, then we might start a new column labeled `RGDP`. We would then compute the value of `RGDP` for the first observation and copy the formula to all of the other observations in that column. Here one line of code computes them all.

Digression. There are two syntax issues we should mention. One is that others -- not us! -- commonly refer to a variable `df[` GDP `]` by `df.GDP`. That usually works, but not always. For example, it doesn't work if the variable name contains spaces or conflicts with an existing method. And we can't assign to it, as we did when we defined `RGDP` above. The second issue is integer variable names. We avoid these, too, but if we somehow end up with a variable with an integer label -- `2011`, for example -- we would refer to it by that label: `df[2011]` without quotes. If you're not sure what type the variable labels are, print `df.columns` and see whether they have quotes.

Rename variables. If we want to change *all the names*, we can assign a list of new names to the DataFrame's column labels. WARNING -- You should be very careful when you do this because unless you're sure of what you're doing then you might rename a variable incorrectly!

```
df.columns = ["cpi", "country", "gdp", "year", "rgdp", "gdp_div_1000"]
```

Here we've simply changed the column names to lower case. There's a clever way to do this with a list comprehension:

```
df.columns = [var.lower() for var in df.columns]
```

The flexibility of string methods and list comprehensions opens up a lot of other possibilities as well. We can also assign new names individually. Suppose we want to give `gdp` a more informative name such as `ngdp` (which stands for nominal gdp). We can do that with the statement

```
df = df.rename(columns={'gdp': 'ngdp'})
```

Note the use of a dictionary that associates the old name (the "key" `gdp`) with the new name (the "value" `ngdp`). If we want to change more than one variable name, we simply add more items to the dictionary. Additionally, it is worth noting that we needed to assign the return of the `rename` function to `df` again -- This is because most of the pandas functions are making copies of our dataframes. You will have to assign the output of a method to a dataframe frequently.

Extract variables. We just saw that commands like `df['ngdp']` "extract" the variable/series `ngdp` from the DataFrame `df` . In other cases, we may want to extract a set of variables and create a smaller DataFrame. This happens a lot when our data has more variables than we need.

We can extract variables by name or number. If by name, we simply put the variable names in a list. If by number, we count (as usual) starting with zero. This code gives us two ways to extract `ngdp` and `rgdp` from `df` :

```
namelist = ['ngdp', 'rgdp']
numlist  = [2, 4]
df_v1 = df[namelist]
df_v2 = df[numlist]
```

You might verify that the two new DataFrames are identical -- with a small dataframe like this, you can do this by hand by just printing them both out.

Closely related is the `drop` method. If we want to drop the variable `cpi` , we would use

```
df.drop(['cpi'], axis=1)
```

Here `axis=1` refers to columns; `axis=0` refers to rows.

Exercise. For the DataFrame `df`, create a variable `diff` equal to the difference of `ngdp` and `rgdp`. Verify that `diff` is now in `df`.

Exercise. How would you extract the variables `ngdp` and `year`?

Exercise. How would you drop the variable `diff`? If you print your dataframe again, is it gone? If not, why do you think it is still there?

Exercise (very challenging). Use a list comprehension to change the variable names from `['ngdp', 'rgdp', 'gdp_div_1000']` to `['nGDP', 'rGDP', 'GDP_div_1000']`. *Hint:* What does `s.replace('gdp', 'GDP')` do to the string `s = "this_is_gdp"`?

DataFrame methods

One of the great things about DataFrames is that they have lots of methods ready to go. We'll survey some of the most useful ones at high speed and come back to them when we have more interesting data.

Data output. To save a DataFrame to a local file on our computer we use the `df.to_*` family of methods. For example, the methods `df.to_csv()` and `df.to_excel()` produce csv and Excel files, respectively. Both require a file name as input. We'll hold off on them until we've addressed files on our computer.

Clipboard methods. We can read from the clipboard and write to it. Suppose we open a spreadsheet and copy a section of it into the clipboard. We can paste it into a DataFrame with the statement

```
df_clip = pd.read_clipboard()
```

Going the other way, we can copy the DataFrame `df` to the clipboard with `df.to_clipboard()`. From the clipboard, we can paste it into Excel or other applications. We're not fans of this -- it makes replication hard if we need to do this again -- but it's awful convenient. We heard about it from one of our former students.

The top and bottom of a DataFrame. We commonly work with much larger DataFrames in which it's unwieldy, and perhaps impossible, to print the whole thing. So we often look at either the top or bottom: the first few few or last few observations. The statement `df.head(n)` extracts the top `n` observations and `df.tail()` (with no input) extracts the bottom 5. This creates a new DataFrame, as we see here:

```
h = df.head(2)
print(type(h))
print(h)
```

The second print statement gives us the first 2 observations, which is what we requested.

`df.tail(2)` does the same for the bottom of the DataFrame `df`: the last 2 observations (rows).

Setting the index. We're not stuck with the index in our DataFrame, we can make it whatever we want. If we want to use `year` as the index, associating observations with the `year` variable, we use the `set_index()` method:

```
df = df.set_index(['year'])
```

That gives us

```
          cpi country     ngdp       rgdp  gdp_div_1000
year
1990  127.500      US   5974.7  46.860392        5.9747
2000  169.300      US  10031.0  59.249852       10.0310
2010  217.488      US  14681.1  67.503035       14.6811
```

with `year` now used as the index.

We did something else here that's important: We assigned the result back to `df`. That keeps what we've done in the DataFrame `df`. If we hadn't done this, `df` would remain unchanged with a counter as its index and our effort to set the index would be lost.

We can reverse what we did here with the `reset_index()` method:

```
df_reset = df.reset_index()
```

Try it and see. We'll spend more time on these methods in a couple weeks.

Statistics. We can compute the mean, the standard deviation, and other statistics for all the variables (this means that they are operating column by column) at once with

```
df.mean()
df.std()
df.describe()
```

The first line gives us the means, the second the standard deviations. The third line gives us a collection of statistics, including the mean, the standard deviation, the min, and the max.

Note that we've done them all at once. `df.mean()` , for example, computes the means of all the variables in one line. Ditto the others.

Notice how pandas was smart and only tried to do compute these statistics for columns with numerical data (e.g. it skipped over the `country` column). This -- pandas doing intelligent things so our code "just works" -- is a theme we'll see many times over the coming weeks.

Plotting. We have a number of methods available that plot DataFrames. The most basic is the `plot()` method, which plots all of the variables against the index. Try this and see what it looks like:

```
df.plot()
```

You should see lines for each of the variables plotted against the index `name` .

Let's put what we've learned to work:

Exercise. Set `year` as the index and assign the result to the DataFrame `dfi` . Use the `index` method to extract it and verify that `year` is, in fact, the index.

Exercise. Apply the `reset_index()` method to our new DataFrame `dfi` . What does it do? What is the index of the new DataFrame?

Exercise. What kind of object is `df.mean()` ?

Exercise. Copy the DataFrame `df` into an empty spreadsheet on your computer using the `to_clipboard()` method.

Exercise. Produce a bar chart of `df` with the statement (Hint: use the docstring for `df.plot` to see what `kind` argument does)

Pandas 2: Data input

Overview. In the last lecture, we introduced some of the main ideas in the pandas package. We now talk about how we can use pandas (and pandas-datareader) to read data into Python. We will learn to read data from both the internet and from our computers.

Python tools. Pandas, Pandas-Datareader, reading spreadsheet files, data

Buzzwords. csv, excel files, datareader, API.

Applications. Income and output of countries, government debt, income by college major, old people, equity returns, George Clooney's movie roles.

Code. Link.

We're finally ready now to look at some data. Lots of data. You will need an **internet connection** for many of our examples. Additionally, you will need to make sure that you have a new package installed -- It is called pandas-datareader. Recall that you can do this by running `conda install pandas-datareader` in your terminal (Refer to the packages chapter if you need a refresher on conda).

In this lecture, our main goal is to teach you some of the functions that we will use to read data (both from your own computer and from the internet).

Recall that our typical program will consist of data input, data management, and graphics creation. The package we previously introduced, pandas, will allow us to do all of these things -- Though eventually as we make more complicated graphics, we will need to use other packages such as: matplotlib, seaborn, plotly, etc... This lecture will focus mostly on data input; we will discuss the others (data management and graphics) in detail in later lectures.

Reminders

- Objects and methods. Recall that we apply the method `justdoit` to the object `x` with `x.justdoit()` .
- Help. We get help in Spyder from both the IPython console and the Object inspector. For the hypothetical `x.justdoit` , we would type `x.justdoit?` in the IPython console or `x.justdoit` in the Object inspector.

- Data structures. That's the term we use for specific organizations of data. Examples are lists, tuples, and dictionaries. Each has a specific structure and a set of methods we can apply. List are (ordered) collections of objects between square brackets: `numberlist = [1, -5, 2]` . Dictionaries are (unordered) pairs of items between curly brackets: `namedict = {'Dave': 'Backus', 'Chase': 'Coleman'}` . The first item in each pair is the "key," the second is the "value.""
- Packages. Packages are collections of tools -- functions and types/methods -- that extend Python's capabilities. We import a package using an `import` (e.g. `import pandas`) statement or some combination of `import` , `from` , and `as` (e.g. `import pandas as pd` or `from pandas import DataFrame`).
- Pandas. Pandas is oriented around two main types: `DataFrame` and `Series` . A `DataFrame` is essentially just a table of data and a `Series` can be thought of as a one columned `DataFrame` . We discussed last time some of the associated properties and methods -- `df.shape` , `df.dtype` , `df.T` , `df.mean()` ...

Data input 1: reading internet files

The easiest way to get data into a Python program is to read it from a file -- a spreadsheet file, for example. The word "read" here means take what's in the file and somehow get it into Python so we can do things with it. Pandas can read many types of files: csv, xls, xlsx, and so on. The files can be on our computer or on the internet. We'll start with the internet -- there's less ambiguity about the location of the file -- but the same approach will work with files on your computer.

We prefer **csv files** ("comma separated values"), a common data format for serious data people. Their simple structure (entries separated by commas) allows easy and rapid input. They also avoid some of the problems with translating Excel files. If we have an Excel spreadsheet, we can always save it as a "CSV (Comma delimited) (*.csv)" file. Excel will warn us that some features are incompatible with the csv format, but we're generally happy to do it anyway. Here's an example of a raw csv file (pretty basic, eh?) and this is (roughly) how it's displayed in Excel.

Reading csv files. It's easy to read csv files with Pandas. We'll read one from our GitHub repository to show how it works. We like to read data from internet sources like this, especially when the data is automatically updated at the source. That's not the case here, but we'll see how easy it is to get this kind of data into Python. We read the cleverly-named `test.csv` with the equally clever `read_csv` function in Pandas:

```
import pandas as pd
url1 = 'https://raw.githubusercontent.com/NYUDataBootcamp'
url2 = '/Materials/master/Data/test.csv'
url  = url1 + url2         # location of file
df = pd.read_csv(url)      # read file and assign it to df
```

The syntax works like this:

- `url` is a string that tells Python where to look for the file. We break it in two because it's too long to fit on one line (remember that we want you to keep your code at 80 characters or less!).
- `read_csv()` is a Pandas function that reads csv files. The `pd.` before it tells Python it's a Pandas function; we established the `pd` abbreviation in the `import` statement.
- The `df` on the left makes this an assignment: We assign what we read to the variable `df`.

So what does the `read_csv` function give us? What's in `df`? We can check its contents by adding the statement `print('\n', df)`. (The `'\n'` tells the print function to start printing on a new line, which makes the output look better.) The result is

```
      name  x1  x2   x3
0     Dave   1   2  3.5
1    Chase   4   3  4.3
2  Spencer   5   6  7.8
```

What we have is a table, much like what we'd see in a spreadsheet. If we compare it to the source we see that the index has been added by the program (just like in the last lecture), but the others are just as they look in the source.

Note that the table of data has labels for both the columns and rows. We'll do more with both of them shortly.

The documentation for `read_csv` in the Object inspector gives us an overwhelming amount of information. Starting at the bottom, we see that it returns a DataFrame. We also see a long list of optional inputs that change how we read the file. Here are some examples we've found useful:

Example. Change the last line of the earlier code to

```
dfalt = pd.read_csv(url, nrows=2)
print('\n', dfalt)
```

The argument `nrows=2` tells the `read_csv` statement to read only the first 2 rows of the file at `url`. One of the instances in which this can be useful if you have a very large dataset and just want a small subset of data to explore while you develop your code.

Example. We can identify specific values in a csv file as missing or NA (not available). We see in the documentation that the parameter `na_values` takes a list of strings as input. To treat the number 1 as missing we change the read statement to `read_csv(url, na_values=[1])`. The result is

```
      name  x1  x2   x3
0     Dave NaN   2  3.5
1    Chase   4   3  4.3
2  Spencer   5   6  7.8
```

We see that the number 1 that was formerly at the top of the `x1` column has been replaced by `NaN` -- "not a number". This particular example is somewhat silly, as we probably do not want to pretend that the number 1 is actually missing data. However, as we will see throughout the course, real data is messy and often has missing values. We've come across all these markers (and more) for noting that some data is missing: `-`, `null`, `n/a`, `NA`, `N/A`, `nan`, `NaN`, ... Being able to tell pandas which values to treat as missing will allow us to read in some otherwise troublesome data.

Reading Excel files. We can also read Excel files (xls and xlsx) with Pandas using the `read_excel()` function. The syntax is almost identical:

```
import pandas as pd
url1 = 'https://raw.githubusercontent.com/NYUDataBootcamp'
url2 = '/Materials/master/Data/test.xls'
url  = url1 + url2
dfx = pd.read_excel(url)
print('\n', dfx)
```

If all goes well, the modified code produces a DataFrame `dfx` that's identical to `df`.

Exercise. Run the code

```
url1 = 'https://raw.githubusercontent.com/NYUDataBootcamp'
url2 = '/Materials/master/Data/test0.csv'    # note the added 0
url  = url1 + url2
df = pd.read_csv(url)
```

What happens? Why? (Hint: Try going to that url in your browser)

Exercise. Delete the `0` in `test0` and rerun the code. What happens if you add the argument `index_col=0` to the `read_csv` statement? How does `df` change?

Exercise. In the `read_excel` code, change the file extension at the end of `url2` from `.xls` to `.xlsx`. What does the new code produce?

Exercise. Adapt the `read_csv` code to treat the numbers 1 and 6 as missing. *Hint:* See the example a page or so back.

Data input 2: Reading files from your computer

Next up: reading files in Python from your computer's hard drive. This is really useful, but there's a catch: we need to tell Python where to find the file.

Prepare test data. We start with the easy part. Open a blank spreadsheet in Excel and enter the data (Pro tip: Remember that we mentioned the `df.to_clipboard()` method? This might be a good time to use it):

```
name    x1  x2   x3
Dave     1   2  3.5
Chase    4   3  4.3
Spencer  5   6  7.8
```

That is: four rows with four entries in each one.

Now save the contents in your `Data_Bootcamp` directory. Do this three times in different formats:

- Excel file. Save the file as `test.xlsx`.
- Old-style Excel file. Save as an "Excel 97-2003 (*.xls)" file under the name `test.xls`.
- CSV file. Save as a "CSV (Comma delimited) (*.csv)" file under the name `test.csv`.

Each of these options shows up in Excel when we choose "Save As."

Find the file. Ok, now where is the file? We know, it's in the `Data_Bootcamp` directory, but where is that? We need the complete path so we can tell Python where to find it.

Exercise (potentially challenging). We can save some time if you already know this. (It's not required, we're just giving it a shot.) Find the path to `test.csv` on your computer. If you find it, tell us what you did. If you're not sure what this means, or how to do it, raise your hand.

Let's introduce some terms so we can be clear what we're talking about. The "file name" is something like `test.csv`. The format of the "directory" or folder address depends on the operating system. On a Windows computer, it's something like

```
C:\Users\userid\Documents\Data_Bootcamp
```

On a Mac, it looks like

```
/Users/userid/Data_Bootcamp
```

The complete path to the file `test.csv` is a combination of the path to the directory and the file name, with a slash in between. In Windows:

```
C:\Users\userid\Documents\Data_Bootcamp\test.csv
```

In Mac OS:

```
/Users/userid/Data_Bootcamp/test.csv
```

Note that they use different kinds of slashes.

How did we find these addresses or paths?

- Windows. Type the name of the file in the Windows search box or use Windows Explorer.
- Mac OS. We select (but not open) the file and do "command-i" to get the file's information window. From the window, we copy the path that follows "Where:." It looks like we're copying arrows, but they turn into slashes when we paste the path.

[Comments welcome on how to make this clearer.]

Reading data with the complete path. Once we have the complete path, we simply tell Python to read the file at that location. Again, this varies with the operating system.

- In Windows, we take the complete path and -- **this is important** -- change all the backslashes `\` to either double backslashes `\\` or forward slashes `/`. (Don't ask.) Then we read the file from the path, just as we read it from a url earlier:

```
path = 'C:\\Users\\userid\\Documents\\Data_Bootcamp\\test.csv'
df = read_csv(path)
```

- In Mac OS we don't need to change the slashes:

```
path = '/Users/userid/Data_Bootcamp/test.csv'
df = read_csv(path)
```

- In both: Open the file in Excel, click on File, and read the path from the Info tab.

Exercise. Read `test.csv` using the path and the method described above. Let us know if you have trouble.

Reading from the current working directory. An alternative is to set the current working directory (cwd), which is where Python will look for files. We can set that with Python's os module. What you'll need is the location of the `Data_Bootcamp` directory.

Once we know the directory path, we can use it in Python.

- In Windows, we use

  ```
  import os

  file = 'test.csv'
  cwd  = 'C:/Users/userid/Data_Bootcamp'

  os.chdir(cwd)                              # set current working directory
  print('Current working directory is', os.getcwd())
  print('File exists?', os.path.isfile(file))   # check to see if file is there

  df = pd.read_csv(file)
  ```

- In Mac OS, the only difference is the format of the path:

  ```
  import os

  file = 'test.csv'
  cwd  = '/Users/userid/Data_Bootcamp'

  os.chdir(cwd)                              # set current working directory
  print('Current working directory is', os.getcwd())
  print('File exists?', os.path.isfile(file))   # check to see if file is there

  df = pd.read_csv(file)
  ```

Once we've set the path, we read the csv file as before. `read_excel()` works the same way with Excel files.

Report problems. If you have difficulty, or find that this works differently on your computer, let us know.

Data input: Examples

Here are some spreadsheet datasets we find interesting. In each one, we describe the data using the `shape`, `columns`, and `head()` methods. Where we can, we also produce a simple plot.

At this point you should **download the code file** for this chapter, linked on the first page (the top of this page if you are viewing online). Save it in your `Data_Bootcamp` directory and open it in Spyder. That will save you a lot of typing.

Penn World Table. The PWT, as we call it, is a standard database for comparing the incomes of countries. It includes annual data for GDP, GDP per person, employment, hours worked, capital, and many other things. The variables are measured on a comparable basis, with GDP measured in 2005 US dollars.

The data is in an Excel spreadsheet. If we open it, we see that it has three sheets. The third one is the data and is named `Data`. We read it in with the code:

```
url = 'http://www.rug.nl/research/ggdc/data/pwt/v81/pwt81.xlsx'
pwt = pd.read_excel(url, sheetname='Data')
```

So what does that give us?

- `pwt.shape` returns `(10357, 47)`: the DataFrame `pwt` contains 10,357 observations of 47 variables.
- `list(pwt)` gives us the variable names, which include `countrycode`, `country`, `year`, `rgdpo` (real GDP), and `pop` (population).
- `pwt.head()` shows us the first 5 observations, which refer to Angola for the years 1950 to 1954. If we look further down, we see that countries are stacked on top of each other in alphabetical order.

In this dataset, each column is a variable and each row is an observation. But if we were to plot one of the variables, it wouldn't make much sense. The observations string together countries, one after the other. What we'd like to do is compare countries, which this isn't set up to do -- yet.

Exercise. Download the spreadsheet and open it in Excel. What does it look like? (You can use your Google fu here: Google "penn world table 8.1", go to the first link, and look for the Excel link.)

Exercise. Change the input in the last line of code to `sheetname=2`. Why does this work?

World Economic Outlook. Another good source of macroeconomic data for countries is the IMF's World Economic Outlook or WEO. It comes out twice a year and includes annual data from 1980 to roughly 5 years in the future (forecasts, evidently). It includes the usual GDP, but also government debt and deficits, interest rates, and exchange rates.

This one gives us some idea of the challenges we face dealing with what looks like ordinary spreadsheet data. The file extension is `xls`, which suggests it's an Excel spreadsheet, but that's a lie. In fact it's a "tab-delimited" file: essentially a csv, but with tabs rather than commas separating entries. We read it with

```
url1 = 'https://www.imf.org/external/pubs/ft/weo/'
url2 = '2015/02/weodata/WEOOct2015all.xls'
weo = pd.read_csv(url1+url2,
                  sep='\t',                # \t = tab
                  thousands=',',           # kill commas
                  na_values=['n/a', '--']) # missing values
```

This has several features we need to deal with:

- Use `read_csv()` rather than `read_excel()`: it's not an Excel file despite what the file name suggests
- Identify tabs as the separator between entries with the argument `sep='\t'`.
- Use the `thousands` argument to eliminate commas from numbers -- things like `12,345.6`, which Python will treat as strings. (What were they thinking of?)
- Identify missing values with the `na_values` argument.

Keep in mind that it took us an hour or two to figure all this out. You can get a sense of where we started by running the `read_csv` statement without the last two arguments and listing its dtypes. You'll notice that variables you might expect to be floats are objects instead.

Exercise. Download the WEO file. What happens when you open it in Excel? (You can use the link in the code. Or Google "imf weo data", look for the most recent link, and choose Entire Dataset.)

Exercise. Why were we able to spread the `read_csv()` statement over several lines?

Exercise. Google "python pandas weo" to see if someone else has figured out how to read this file.

Exercise. How big is the DataFrame `weo`? What variables does it include? Use the statement `weo[[0, 1, 2, 3, 4]].head()` to see what the first five columns contain.

This dataset doesn't come in the standard format, with columns as variables and rows as observations. Instead, each row contains observations for all years for some variable and country combination. If we want to work with it, we'll have to change the structure. Which we'll do, but not now.

PISA education data. PISA stands for Program for International Student Assessment. It's an international effort to collect information about educational performance that's comparable across countries. We read about it every few years when newspapers print stories about how poorly American students are doing. PISA collects data on student test performance, teacher quality, and many other things, and posts both summaries and individual test results.

We use data from a summary table in an OECD report; note the data link at the bottom of Table 1.A. This code reads the data from the link:

```
url = 'http://dx.doi.org/10.1787/888932937035'
pisa = pd.read_excel(url,
                     skiprows=18,             # skip the first 18 rows
                     skipfooter=7,            # skip the last 7
                     parse_cols=[0,1,9,13],   # select columns of interest
                     index_col=0,             # set the index as the first column
                     header=[0,1]             # set the variable names
                     )
```

There are a number of new things in the read statement:

- We spread the read statement over several lines to make it easier to read. Python understands that the line doesn't end until we reach the right paren `)`. That's common Python syntax.
- We skip rows at the top and bottom that do not contain data.
- We choose specific columns to read using the `parse_cols` parameter. Column numbering starts at zero, as we have come to expect. Here we select the mean score for each field (math, reading, science).
- We set the index as the first column.
- We set the header from the first two rows (after those we skip). Note that the header has two parts: the field (math, reading, science) and the measurement (mean, share of low-achievers, etc).

We can clean this up further if we drop blank lines and simplify the variable names:

```
pisa = pisa.dropna()                               # drop blank lines
pisa.columns = ['Math', 'Reading', 'Science']    # simplify variable names
pisa['Math'].plot(kind='barh')
```

The plot we produce in the last line is virtually impossible to read, but we'll work on that later.

UN population data. We tend to have pretty good demographic data. We keep track of how many people we have, their ages, how many children they have, what they die of, and so on. A good international source is the United Nations' Population Division.

This code reads in estimates of population by age for many countries:

```
url1 = 'http://esa.un.org/unpd/wpp/DVD/Files/'
url2 = '1_Indicators%20(Standard)/EXCEL_FILES/1_Population/'
url3 = 'WPP2015_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES.XLS'
url = url1 + url2 + url3

cols = [2, 4, 5] + list(range(6,28))
est = pd.read_excel(url, sheetname=0, skiprows=16, parse_cols=cols)
```

The columns contain population numbers for 5-year age groups. When we're up to it, we'll use this data to illustrate the dramatic aging of the population in many countries. It's one of the striking facts of modern times: people are living longer, a lot longer.

Exercise. What does `list(range(6,28))` do? Why?

Incomes by college major. Nate Silver's 538 blog does a lot of good data journalism and often posts its data online. This one comes from their analysis of income by college major. The data comes from the American Community Survey but they've done the work of organizing it for us.

Here's the code:

```
url1 = 'https://raw.githubusercontent.com/fivethirtyeight/data/master/'
url2 = 'college-majors/recent-grads.csv'
url = url1 + url2
df538 = pd.read_csv(url)
```

Exercise. What variables does this data contain?

Exercise. Set the index as `Major`. (Ask yourself: What method should I use?)

Exercise. Create a horizontal bar chart with the variable `Median` (median salary) using the `plot()` method.

Internet Movie Database (IMDb). We love this one, a list of roles in IMDb's movie database we got from Brandon Rhodes. We read it with this code:

```
url  = 'http://pages.stern.nyu.edu/~dbackus/Data/cast.csv'
cast = pd.read_csv(url, encoding='utf-8')
```

Don't panic if nothing happens for a while. It's a big file (approximately 200 MB) and takes several minutes to read.

Exercise. Since we're all experts by now, we'll leave this one to you:

- How large is the DataFrame?
- What variables does it include?
- Try these statements:

```
ah = cast[cast['title'] == 'Annie Hall']
gc = cast[cast['name'] == 'George Clooney']
```

This goes beyond what we've done so far, but what do you think they do? What do the DataFrames `ah` and `gc` contain?

Data input 3: APIs

APIs are "application program interfaces". That's a mouthful. A dataset with an API allows access through some method other than a spreadsheet. The API is the set of rules for accessing the data. The bad news is the jargon. The good news is that people have written easy-to-use code to access the APIs. We don't need to understand the API, we just use the code -- and say thank you!

The Pandas developers have created what they call a set of Remote Data Access tools and have put them into a package called `pandas_datareader`. These break now and then, typically when the underlying data changes, but when they work they're great.

FRED. The St Louis Fed has put together a large collection of time series data that they refer to as FRED: Federal Reserve Economic Data. They started with the US, but now include data for many countries.

The Pandas docs describe how to access FRED. Here's an example that reads in quarterly data for US real GDP and real consumption and produces a simple plot:

```
from pandas_datareader import data, wb
import datetime                  # package to handle dates

start = datetime.datetime(2010, 1, 1)  # start date
codes = ['GDPC1', 'PCECC96']     # real GDP, real consumption
fred  = data.DataReader(codes, 'fred', start)
fred = fred/1000                 # convert billions to trillions

fred.plot()
```

We copied most of this from the Pandas documentation. Which is a good idea: Start with something that's supposed to work and change one thing at a time until you have what you want.

The variable `start` contains a date in (year, month, day) format. Pandas knows a lot about how to work with dates, especially when we construct them using `datetime.datetime` as we did above. We're hoping to cover pandas' time series capabilities in depth in a future chapter of the book.

The variable `codes` -- not to be confused with "code" -- consists of FRED variable codes. Go to FRED, use the search box to find the series you want, and look for the variable code at the end of the url in your browser.

Exercise. Run the same code with a start date of 2005. What do you see?

World Bank. The World Bank's databank covers economic and social statistics for most countries in the world. Variables include GDP, population, education, and infrastructure. Here's an example:

```
from pandas_datareader import data, wb

var = ['NY.GDP.PCAP.PP.KD']         # GDP per capita
iso = ['USA', 'FRA', 'JPN', 'CHN', 'IND', 'BRA', 'MEX']  # country codes
year = 2013
wbdf = wb.download(indicator=var, country=iso, start=year, end=year)
```

If we look at the DataFrame `wbdf`, we see that it has a double index, `country` and `year`. By design, all of the data is for 2013, so we kill off that index with the `reset_index` method and plot what's left as a horizontal bar chart:

```
wbdf = wbdf.reset_index(level='year', drop=True)
wbdf.plot(kind='barh')
```

(Trust us on the `drop=True`. We'll come back to it in a couple weeks.)

We use codes here for countries and variables. We can find country codes in this list -- or just Google "country codes". Pandas accepts both 2- and 3-letter versions. We find variable codes with the search tool in the Remote Data Access module or by looking through the World Bank's data portal. We prefer the latter. Click on a variable of interest and read the code from the end of the url.

Exercise. What would you like to change in this graph? Keep a list for the next time we run into this one.

Fama-French. Gene Fama and Ken French post lots of data on equity returns on Ken French's website. The data are zipped text files, which we can easily read into Excel. The Pandas tool is even better. Here's an example:

```
from pandas_datareader import data, wb

ff = data.DataReader('F-F_Research_Data_factors', 'famafrench')[0]
ff.columns = ['xsm', 'smb', 'hml', 'rf']      # rename variables

ff.describe()
```

The data is monthly, 1926 to present. Returns are expressed as percentages; multiply by 12 to get annualized returns. The variable names refer to

- `xsm` : the return on the market minus the riskfree return
- `smb` : the return on small firms minus the return on big firms
- `hml` : the return on value firms minus the return on growth firms
- `rf` : the riskfree rate

We use the `describe()` method to compute statistics. Evidently `xsm` has the largest mean. It also has the largest standard deviation. A couple plot methods show us more about the distribution:

```
ff.boxplot()
ff.plot()
ff.plot(ff['xsm'], ff['smb'], kind='scatter')
```

What do you see? What more would you like to know?

Review

Run this code to create a DataFrame of technology indicators from the World Bank for four African countries:

```
import pandas as pd
data = {'EG.ELC.ACCS.ZS': [53.2, 47.3, 85.4, 22.1],     # access to elec (%)
        'IT.CEL.SETS.P2': [153.8, 95.0, 130.6, 74.8],  # cell contracts per 100
        'IT.NET.USER.P2': [11.5, 12.9, 41.0, 13.5],     # internet access (%)
        'Country': ['Botswana', 'Namibia', 'South Africa', 'Zambia']}
af = pd.DataFrame(data)
```

(You can cut and paste this from the bottom of this chapter's code file.)

Exercise. What type of object is `af`? What are its dimensions?

Exercise. What is the index? Change it to country names.

Exercise. What are the variable names? Change them to something more informative.

Exercise. Create a horizontal bar chart with this DataFrame. What does it tell us? Which country has the most access to electricity? Cell phones?

Resources

We've covered a lot of ground, but if you're looking for more we suggest:

- On Pandas: Chris Moffitt's Practical Business Python blog has a good series on Pandas from the perspective of an Excel user. For a more concise summary, try Quandl's cheatsheet.
- On data: See the list of data sources on the course website.
- On backslashes (and other obscure characters in code): xkcd.

Python graphics: Matplotlib fundamentals

Overview. We introduce and apply Python's popular graphics package, Matplotlib. We produce line plots, bar charts, scatterplots, and more. We do all this in Jupyter using a Jupyter notebook.

Python tools. Jupyter notebooks. Graphing with Matplotlib: dataframe plot methods, the `plot(x,y)` function, figure and axis objects.

Buzzwords. Data visualization, Jupyter notebook

Applications. US GDP, GDP per capita and life expectancy, Fama-French asset returns, PISA math scores.

Code. Link.

Computer graphics are one of the great advances of the modern world. Graphs have always been helpful in describing data or concepts, and now they're a lot easier to produce. We've gotten so good at drawing pictures that we invented a new term for it: **visualization**. Done well, a graph tells us something new -- and gets us thinking about other things we'd like to know.

That's the good news. The bad news is that graphics are inherently complicated. Programs like Excel do their best to hide this fact, but if you ever try to customize a chart it quickly rears its ugly head. Have you ever spent a couple hours trying to fine-tune an Excel graph? More? The problem is that even simple graphs have lots of moving parts: the type (line, bar, scatter, etc); the color and thickness of lines, bars, or markers; title and axis labels; their location, fonts, and font sizes; tick marks (location, size); background color; grid lines (on or off); and so on. That's not an Excel problem, it's a problem with graphics in general.

Our goal here is to produce graphs with **Matplotlib**, Python's leading graphics package. There's a lot here, but don't panic, that's the nature of graphics. And it gets easier with experience.

One more thing before we start: **Save the Jupyter notebook** at the Code link above in your `Data_Bootcamp` directory/folder. The link goes to a display of the notebook; you need to click on the Raw button to get the real file. Be sure to download it as filetype ipynb.

Reminders

- Packages. Collections of tools that extend Python's capabilities. We add them with `import` statements.
- Pandas. Python's data management package. We typically add it to our programs with

  ```
  import pandas as pd
  ```

- Objects and methods. Recall -- again! -- that we apply the method `justdoit()` to the object `x` with `x.justdoit()`.
- Dataframe. A data structure like a spreadsheet that includes a table of data plus row and column labels. Typically columns are variables and rows are observations. We get column labels for a dataframe `df` with `df.columns` and row labels with `df.index`.
- Series. We express a single variable `x` in a dataframe `df` as `df['x']`, a series.
- Reading spreadsheets. We "read" spreadsheet data into Python with the `read_csv()` and `read_excel()` functions in Pandas.
- Jupyter. A Python environment in which we create notebooks. These notebooks combine Python code with text and output, including graphics. It's the ideal medium for this topic.

Jupyter notebooks

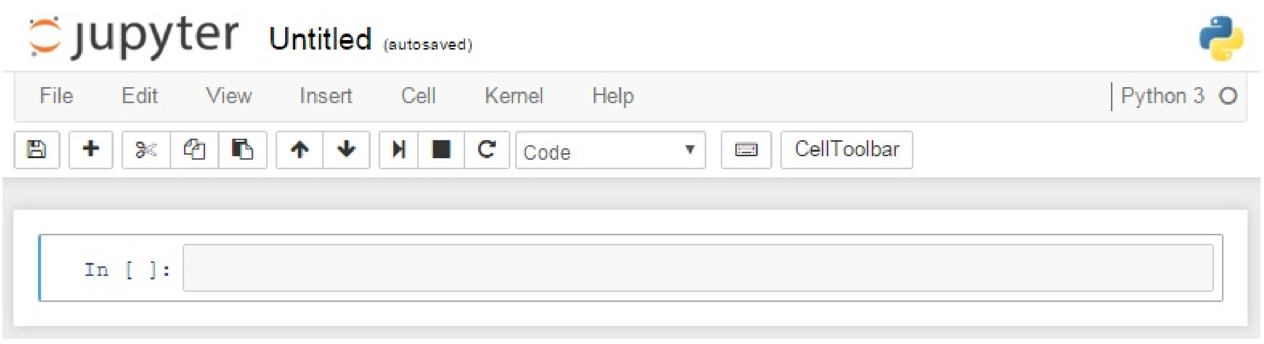
We're going to change programming environments from Spyder to **Jupyter** and work with **Jupyter notebooks**. We had a brief introduction with Jupyter when we installed Anaconda, but we'll go through it again to make sure we're all on the same page.

We can open a new Jupyter notebook by tracing the steps we took in the first class:

- Open your terminal (command prompt on Windows)
- Type `jupyter notebook` and press enter. This will open a tab in your browser with the word Jupyter at the top and your computer's directory structure below it.
- In the browser tab, navigate to your `Data_Bootcamp` directory/folder.
- Click on the New button in the upper right and choose `Python 3` (it may also refer to this as `Python[Root]`.

We now have an empty Jupyter notebook we play with.

Jupyter essentials. In your browser, you should have an empty notebook with the word Jupyter at the top. Below it is a **menubar** with the words File, Edit, View, Cell, Kernel, and Help. Below that is a **toolbar** with various buttons. You can see all of these components here:



If you have a few minutes, click on Help in the menubar and choose User Interface Tour.

Let's put some of these tools to work:

- Change the notebook name. Click on the name (`Untitled` if we just created a new notebook) to the right of the word Jupyter at the top. A textbox should open up. Use it to change the name to `bootcamp_sandbox` .
- Toolbar buttons. Let your mouse hover over one of them to see what it does.
- Add a cell. Click on the `+` in the toolbar to create a new cell. Choose Code in the toolbar's dropdown menu. Type this code in the cell:

  ```
  import datetime as dt
  print('Welcome to Data Bootcamp!')
  print('Today is: ', dt.date.today())
  ```

  Now click on Cell in the menubar and choose Run cell. You should see the welcome message and today's date below the code.
- Add another cell. Click on the `+` to create another cell and choose Markdown in the toolbar's dropdown menu. Markdown is text; more on it shortly. Type this in the cell:

  ```
  Your name
  Data Bootcamp sandbox for playing around with Jupyter notebooks
  ```

  Run this cell as well.

You get the idea. To get a sense of what's possible, take a look at these two notebooks 1 2.

More than you need In addition to the buttons near the top of your notebook, there are also keyboard shortcuts for all these commands. We'll tell you about them along the way. Once we got used to them, we found that the keyboard shortcuts are an easier and more efficient way to do what we need. These will always be noted with **mtyn**. The command for creating a new cell is to press escape to be in *command mode* and then press `a` to insert a new cell above the current one and `b` to insert a new cell below the current cell.

Markdown essentials. Markdown is a simplified version of html ("hypertext markup language"), the language used to construct basic websites. html was a great thing in 1995, but now that the excitement has warn off we find it painful. Markdown, however, has a zen-like simplicity and beauty. Here are some things we can do with it:

- Headings. Large bold headings are marked by hashes (`#`). One hash for first level (very large), two for second level (a little smaller), three for third level (smaller still), four for fourth (the smallest). Try these in a Markdown cell to see how they look:

  ```
  # Data Bootcamp sandbox
  ## Data Bootcamp sandbox
  ### Data Bootcamp sandbox
  ```

  Be sure to run the cell when you're done (`shift enter`).

- Bold and italics. If we put a word or phrase between double asterisks, it's displayed in bold. Thus `**bold**` displays as **bold**. If we use single asterisks, we get italics: `*italics*` displays as *italics*.

- Bullet lists. If we want a list of items marked by bullets, we start with a blank line and mark each item with an asterisk on a new line:

  ```
  * something
  * something else
  ```

  Try it and see.

- Links. We construct a link with the text in square brackets and the url in parentheses immediately afterwards. Try this one:

  ```
  [Data Bootcamp course](http://nyu.data-bootcamp.com/)
  ```

We can find more information about Markdown under Help. Or use your Google fu. We like the Daring Fireball description.

Markdown is ubiquitous. This book, for example, is written in Markdown. Look here for a list of chapter files. Click on one to see how it displays. Click on the Raw button at the top to see the Markdown file that produced it.

Jupyter help. We can access documentation just as we did in Spyder's IPython console: Type a function or method, add a question mark, and run the cell (`shift enter`). For example: `print?` or `df.plot?` .

Exercise. Create a description cell in Markdown at the top of your notebook. It should include your name and a description of what you're doing in the notebook. For example: "Joan Watson's notes on the Data Bootcamp Matplotlib notebook" and a date. *Bonus points:* Add a link.

Exercise. Add two new cells. In the first one, add the statement `import pandas as pd` , labelled as code. Run it. Use the second cell to find documentation for `pd.read_csv` .

Getting ready

We need to do a few things before we're ready to produce graphs.

Open the graphics notebook. If you followed instructions -- and we're confident you did -- you saved the notebook for this chapter in your `Data_Bootcamp` directory. Return to the Jupyter tab in your browser that points to that directory. Look for the file named `bootcamp_graphics.ipynb` . Click to open it. That will open the notebook in a new tab. The notebook will say at the top: "Python graphics: Matplotlib fundamentals" in large bold letters.

Import packages. We need to tell our program what packages we plan to use. The following code also checks their versions and prints the date:

```
import sys                                      # system module
import pandas as pd                             # data package
import matplotlib as mpl                        # graphics package
import matplotlib.pyplot as plt                  # pyplot module
import datetime as dt                           # date and time module

# check versions (overkill, but why not?)
print('Python version:', sys.version)
print('Pandas version: ', pd.__version__)
print('Matplotlib version: ', mpl.__version__)
print('Today: ', dt.date.today())
```

All of these statements generally go at the top of our program -- right after the description.

Process data. We use three dataframes to illustrate Matplotlib graphics.

US GDP. The first one is several years of US GDP and Consumption. We got the numbers from FRED, but have written them out here for simplicity. The code is

```
gdp  = [13271.1, 13773.5, 14234.2, 14613.8, 14873.7, 14830.4, 14418.7,
        14783.8, 15020.6, 15369.2, 15710.3]
pce  = [8867.6, 9208.2, 9531.8, 9821.7, 10041.6, 10007.2, 9847.0, 10036.3,
        10263.5, 10449.7, 10699.7]
year = list(range(2003,2014))        # use range for years 2003-2013

# Note that we set the index
us = pd.DataFrame({'gdp': gdp, 'pce': pce}, index=year)
print(us)
```

Note that we created a dataframe from a dictionary. That's convenient here, but in most real applications we'll read in spreadsheets or access the data online through an "API".

World Bank. Our second dataframe contains 2013 data for GDP per capita (basically income per person) for several countries:

```
code    = ['USA', 'FRA', 'JPN', 'CHN', 'IND', 'BRA', 'MEX']
country = ['United States', 'France', 'Japan', 'China', 'India',
             'Brazil', 'Mexico']
gdppc   = [53.1, 36.9, 36.3, 11.9, 5.4, 15.0, 16.5]

wbdf = pd.DataFrame({'gdppc': gdppc, 'country': country}, index=code)
wbdf
```

In a notebook, the last line -- the dataframe name `wbdf` on its own -- results in the display of `wbdf`. That works as long as it's the last statement in the cell.

Fama-French returns. Our third dataframe consist of annual returns from our friends Fama and French:

```
import pandas.io.data as web
ff = web.DataReader('F-F_Research_Data_factors', 'famafrench')[1]
ff.columns = ['xsm', 'smb', 'hml', 'rf']
ff['rm'] = ff['xsm'] + ff['rf']
ff = ff[['rm', 'rf']]               # extract rm (market) and rf (riskfree)
ff.head(5)
```

This gives us a dataframe with two variables: `rm` is the return on the equity market overall and `rf` is the riskfree return.

Exercise. What kind of object is `wbdf`? What are its column and row labels?

Exercise. What is `ff.index`? What does that tell us?

Digression: Graphing in Excel

Before charging ahead, let's review how we would create what Excel calls a "chart". We need to choose:

- Data. We would highlight a block of cells in a spreadsheet.
- Chart type. Lines, bars, scatter plots, and so on.
- `x` and `y` variables. Typically we graph some `y` variable -- or perhaps several of them -- against an `x` variable, with `x` on the horizontal axis and `y` on the vertical axis. We need to tell Excel which is which.

This might be followed by a long list of fine-tuning: what the lines look like, how the axes are labeled, and so on. We'll see the same in Matplotlib.

Three approaches to graphics in Matplotlib

Back to graphics. Python's leading graphics package is **Matplotlib**. Matplotlib can be used in a number of different ways:

- Approach #1: Apply plot methods to dataframes.
- Approach #2: Use the `plot(x,y)` function to plot `y` against `x`.
- Approach #3: Create figure objects and apply methods to them.

They call on similar functionality, but use different syntax to get it.

Approach #1: Apply plot methods to dataframes

The simplest way to produce graphics from a dataframe is to apply a plot method to it. Simple is good, we do this a lot.

If we compare this to Excel, we will see that a number of things are preset for us:

- Data. By default (meaning, if we don't do anything to change it) the data consists of the whole dataframe.
- Chart type. We'll see below that we have options for lines, bars, or other things.
- `x` and `y` variables. By default, the `x` variable is the dataframe's index and the `y` variables are the columns of the dataframe -- all of them that can be plotted (e.g. columns with a numeric dtype).

We can change all of these things, just as we can in Excel, but that's the starting point.

Example (line plot). Enter the statement `us.plot()` into a code cell and run it. This plots every column of the dataframe `us` as a line against the index, the year of the observation. The lines have different colors. We didn't ask for this, it's built in. A legend associates each variable name with a line color. This is also built in.

Example (single line plot). We just plotted all the variables -- all two of them -- in the dataframe `us`. To plot one line, we apply the same method to a single variable -- a series. The statement `us['gdp'].plot()` plots GDP alone. The first part -- `us['gdp']` -- is the single variable GDP. The second part -- `.plot()` -- plots it.

Example (single line plot 2). In addition to getting a series from our dataframe and then plotting the series, we could also set the `y` argument when we call the plot method. The statement `us.plot(y="gdp")` will produce the same plot as `us['gdp'].plot()`.

Example (bar chart). The statement `us.plot(kind='bar')` produces a bar chart of the same data.

Example (scatter plot). In a scatter plot we need to be explicit about `x` and `y`. We'll use `gdp` as `x` and `pce` (consumption) as `y`. The general syntax for a dataframe `df` is `df.plot.scatter(x,y)`. In this case we use

```
us.plot.scatter('gdp', 'pce')
```

The scatter here is not far from a straight line; evidently consumption and GDP go up and down together.

Exercise. Enter `us.plot(kind='bar')` and `us.plot.bar()` in separate cells. Show that they produce the same bar chart.

Exercise. Add each of these arguments, one at a time, to `us.plot()`:

- `kind='area'`
- `subplots=True`
- `sharey=True`
- `figsize=(3,6)`
- `ylim=(0,16000)`

What do they do?

Exercise. Type `us.plot?` in a new cell. Run the cell (shift-enter or click on the run cell icon). What options do you see for the `kind=` argument? Which ones have we tried? What are the other ones?

We can do similar things with the Fama-French dataframe `ff`. The basic plot statement is

```
ff.plot()
```

This has one series (the equity market return `rm`) that varies a lot and one (the riskfree return `rf`) that does not.

Let's think about the returns a little. What does the data tell us about them? That's an easier question to answer if we use a different plot. We like histograms because they describe all the outcomes in a convenient form. Try this code:

```
ff.plot(kind='hist',          # histogram
        bins=20,              # 20 bins
        subplots=True)        # two separate subplots
```

It produces separate histograms of the two variables with 20 "bins" in each, as noted in the comments.

Exercise. Let's see if we can dress up the histogram a little. Try adding, one at a time, the arguments `title='Fama-French returns'`, `grid=True`, and `legend=False`. What does the documentation say about them? What do they do?

Exercise. What do the histograms tell us about the two returns? How do they differ?

Exercise. Use the World Bank dataframe `wbdf` to create a bar chart of GDP per capita, the variable `'gdppc'`. *Bonus points:* Create a horizontal bar chart. Which do you prefer?

Approach #2: `plot(x,y)`

Next up: the popular `plot(x,y)` function from the pyplot module of Matplotlib. We never use this and will go over it at high speed -- or perhaps not at all.

We import the pyplot module with

```
import matplotlib.pyplot as plt
```

This is a more explicit version of Matplotlib graphics in which we specify the `x` and `y` variables directly, much as we did earlier with a scatter plot. A typical statement has the form

```
plt.plot(x, y)
```

The `plt.` prefix identifies `plot()` as a pyplot function. This produces the same kinds of figures we saw earlier, but we get there by a different route.

Here are some examples. To plot GDP on its own, we use the code

```
plt.plot(us.index, us['gdp'])
```

Remind yourself what the `x` and `y` variables are here. With a dataframe plot method, x is automatically the index. Here we must be explicit about it.

If we want two variables in the same graph, we simply add another line:

```
plt.plot(us.index, us['gdp'])
plt.plot(us.index, us['pce'])
```

(In Jupyter, both of these statements must be in the same cell for the lines to show up in the same figure.)

If we want a bar chart we use

```
plt.bar(us.index, us['gdp'])
```

The bars here are off center, so we typically include the argument `align='center'` .

Exercise. Experiment with

```
plt.bar(us.index, us['gdp'],
        align='center',
        alpha=0.65,
        color='red',
        edgecolor='green')
```

Describe what each of these arguments/parameters does.

Approach #3: Create figure objects and apply methods

This approach was mysterious to us at first, but it's now our favorite. The idea is to generate an object -- two objects, in fact -- and apply methods to them to produce the various elements of a graph: the data, their axes, their labels, and so on.

We do this -- as usual -- one step at a time.

Create objects. We'll see these two lines over and over:

```
import matplotlib.pyplot as plt  # import pyplot module
fig, ax = plt.subplots()         # create fig and ax objects
```

Note that we're using the pyplot function `subplots()`, which creates the objects `fig` and `ax` on the left. The `subplot()` function produces a blank figure, which is displayed in the Jupyter notebook. The names `fig` and `ax` can be anything, but these choices are standard.

We say `fig` is a **figure object** and `ax` is an **axis object**. (Try `type(fig)` and `type(ax)` to see why.) Once more, the words don't mean what we might think they mean:

- `fig` is a blank canvas for creating a figure.
- `ax` is everything in it: axes, labels, lines or bars, legend, and so on.

Once we have the objects, we apply methods to them to create graphs.

Create graphs. We create graphs by applying plot-like methods to `ax`. We typically do this with dataframe plot methods:

```
fig, axe = plt.subplots()        # create axis object axe
us.plot(ax=axe)                  # ax= looks for axis object, axe is it
```

(Note again that we need to create and use the axis object in the same code cell.)

Example. Let's do the same with the Fama-French data:

```
fig, ax = plt.subplots()
ff.plot(ax=ax,
        kind='line',                 # line plot
        color=['blue', 'magenta'],   # line color
        title='Fama-French market and riskfree returns')
```

Exercise. Let's see if we can teach ourselves the rest:

- Add the argument `kind='bar'` to convert this into a bar chart.
- Add the argument `alpha=0.65` to the bar chart. What does it do?
- What would you change in the bar chart to make it look better? Use the help facility to find options that might help. Which ones appeal to you?

Exercise (somewhat challenging). Use the same approach to reproduce our earlier histograms of the Fama-French series.

Let's review

Take a deep breath. We've covered a lot of ground, it's time to recapitulate.

We looked at three ways to use Matplotlib:

- Approach #1: Apply plot methods to dataframes.
- Approach #2: Use the `plot(x,y)` function.
- Approach #3: Create `fig, ax` objects and apply plot methods to them.

This is what their syntax looks like applied to US GDP:

```
us['gdp'].plot()                  # Approach #1

plt.plot(us.index, us['gdp'])      # Approach #2

fig, ax = plt.subplots()          # Approach #3
us['gdp'].plot(ax=ax)
```

Each one produces the same graph.

Which one should we use? **Use Approach #3.** Really. This is a case where choice is confusing.

We also suggest you not commit any of this to memory. If you use end up using it a lot, you'll remember it. If you don't, it's not worth remembering. We typically start with examples anyway rather than creating new graphs from scratch.

Bells and whistles

We now know how to create graphs, but if we're honest with ourselves we'd admit they're a little basic. Fortunately, we just got started. We have a huge number of methods available for changing our plots in any way we wish: Add titles and axis labels, change axis limits, and many other things that haven't crossed our minds yet. Here's a short introduction.

Adding things to graphs. So far we've added things to our graph with arguments. Axis methods offer us a lot more flexibility. Consider these:

```
fig, ax = plt.subplots()

us.plot(ax=ax)
ax.set_title('US GDP and Consumption', fontsize=14, loc='left')
ax.set_ylabel('Billions of 2013 USD')
ax.legend(['GDP', 'Consumption'])           # more descriptive variable names
ax.set_xlim(2002.5, 2013.5)                 # shrink x axis limits
ax.tick_params(labelcolor='red')            # change tick labels to red
```

In this way we add a title (14-point type, left justified), add a label to the y axis, change the limits of the x axis, make the tick labels red, and use more descriptive names in the legend. The tick labels, in particular, are extremely ugly, but they illustrate the control we have over figures.

Exercise. Use the `set_xlabel()` method to add an x-axis label. What would you choose? Or would you prefer to leave it empty?

Exercise. Enter `ax.legend?` to access the documentation for the `legend` method. What options appeal to you?

Exercise. Change the line width to 2 and the line colors to blue and magenta. *Hint:* Use `us.plot?` to get the documentation.

Exercise (challenging). Use the `set_ylim()` method to start the `y` axis at zero. *Hint:* Use `ax.set_ylim?` to get the documentation.

Exercise. Create a line plot for the Fama-French dataframe `ff` that includes both returns. *Bonus points:* Add a title with the `set_title` method.

Multiple plots. We've produced, for the most part, single plots. But the same tools can produce multiple plots in one figure.

Here's an example that produces separate "subplots" of US GDP and consumption. We start by creating the objects:

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
print('Object ax has dimension', len(ax))
```

The `subplot` statement asks for a graph with two rows (top and bottom) and one column. That is, two graphs, one on top of the other. The `sharex=True` argument makes the `x` axes the same. The `print` statement tells us "Object ax has dimension 2", one for the GDP graph, and one for the consumption graph.

Now do the same with content:

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)

us['gdp'].plot(ax=ax[0], color='green')   # first plot
us['pce'].plot(ax=ax[1], color='red')     # second plot
```

(Note that we start numbering the components of `ax` at zero, which should be getting familiar by now.) This gives us a double graph, with GDP at the top and consumption at the bottom. Put another way, the figure `fig` contains two axis (`ax[0]` and `ax[1]`) and each axis has one plot in it.

Examples

We conclude with examples that take data from the previous chapter and make better graphs than we did there.

PISA test scores. Recall that we had a simple plot, but it didn't look very good. The code was

```
import pandas as pd
import matplotlib.pyplot as plt

url = 'http://dx.doi.org/10.1787/888932937035'
pisa = pd.read_excel(url,
                     skiprows=18,      # skip the first 18 rows
                     skipfooter=7,     # skip the last 7
                     parse_cols=[0,1,9,13], # select columns of interest
                     index_col=0,      # set the index as the first column
                     header=[0,1]      # set the variable names
                     )
pisa = pisa.dropna()                   # drop blank lines
pisa.columns = ['Math', 'Reading', 'Science'] # simplify variable names

fig, ax = plt.subplots()
pisa['Math'].plot(kind='barh', ax=ax)  # create bar chart
```

Comment. Yikes! That's horrible! What can we do about it? Any suggestions?

The problem seems to be that the bars and labels are squeezed together, so perhaps we should make the figure taller. We set the figure's dimensions with the argument `figsize=(width, height)`. The sizes are measured in inches, which get shrunk a bit when we display them in Jupyter. Here's a version with a much larger `height` that we discovered by experimenting:

```
fig, ax = subplots()
pisa['Math'].plot(kind='barh', ax=ax, figsize=(4,13))
ax.set_title('PISA Math Score', loc='left')
```

This creates a figure that is 4 inches wide and 13 inches tall. We added a title, too, to be clear about what we have. The title has a fontsize of 14 and is left justified.

Here's a more advanced version in which we made the US bar red. This is ridiculously complicated, but we used our Google fu and found a solution. (Remember: The solution to many programming problems is a combination of Google fu and patience.) The code is

```
fig, ax = plt.subplots()
pisa['Math'].plot(ax=ax, kind='barh', figsize=(4,13))
ax.set_title('PISA Math Score', loc='left')
ax.get_children()[36].set_color('r')
```

The `36` comes from experimenting. We count from the bottom starting with zero.

World Bank data. Our second example comes from using the World Bank's API, which gives us access to a huge amount of data for countries. We use it to produce two kinds of graphs and illustrate some tools we haven't seen yet:

- Bar charts of GDP and GDP per capita
- Scatter plot (bubble plot) of life expectancy v GDP per capita

We start with the data:

```
# load packages (redundancy is ok)
import pandas as pd                   # data management tools
from pandas.io import wb              # World Bank api
import matplotlib.pyplot as plt       # plotting tools

# variable list (GDP, GDP per capita, life expectancy)
var = ['NY.GDP.PCAP.PP.KD', 'NY.GDP.MKTP.PP.KD', 'SP.DYN.LE00.IN']
# country list (ISO codes)
iso = ['USA', 'FRA', 'JPN', 'CHN', 'IND', 'BRA', 'MEX']
year = 2013

# get data from World Bank
df = wb.download(indicator=var, country=iso, start=year, end=year)

# munge data
df = df.reset_index(level='year', drop=True)
df.columns = ['gdppc', 'gdp', 'life'] # rename variables
df['pop']  = df['gdp']/df['gdppc']    # population
df['gdp'] = df['gdp']/10**12          # convert to trillions
df['gdppc'] = df['gdppc']/10**3       # convert to thousands
df['order'] = [5, 3, 1, 4, 2, 6, 0]   # reorder countries
df = df.sort_values(by='order', ascending=False)
df
```

Note that the index here is the country name -- that will be our x axis.

Here's a horizontal bar chart for (total) GDP:

```
fig, ax = plt.subplots()
df['gdp'].plot(ax=ax, kind='barh', alpha=0.5)
ax.set_title('GDP', loc='left', fontsize=14)
ax.set_xlabel('Trillions of US Dollars')
ax.set_ylabel('')
```

What do you see? What's the takeaway?

We think the horizontal bar chart looks better than the usual vertical bar chart, which we'd get if we replaced `barh` above with `bar`. (Try it and see what you think.)

Here's a similar chart for GDP per capita:

```
fig, ax = plt.subplots()
df['gdppc'].plot(ax=ax, kind='barh', color='m', alpha=0.5)
ax.set_title('GDP Per Capita', loc='left', fontsize=14)
ax.set_xlabel('Thousands of US Dollars')
ax.set_ylabel('')
```

What do you see here? What's the takeway?

And just because it's fun, here's an example of Tufte-like axes from Matplotlib examples:

```
fig, ax = plt.subplots()
df['gdppc'].plot(ax=ax, kind='barh', color='b', alpha=0.5)
ax.set_title('GDP Per Capita', loc='left', fontsize=14)
ax.set_xlabel('Thousands of US Dollars')
ax.set_ylabel('')

# Tufte-like axes
ax.spines['left'].set_position(('outward', 10))
ax.spines['bottom'].set_position(('outward', 10))
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
```

This gives us axes on the left and bottom only, separated slightly from the bars. It's another illustration of the benefits of Google fu.

We finish off with a bubble plot: a scatter plot in which the size of the dots ("bubbles") varies with a third variable. (Count them: we have `x` on the horizontal axis, `y` on the vertical axis, and a third variable represented by the size of the bubble.) From a technical perspective, this is simply another argument in a scatter plot. Here's an example in which `x` is GDP per capita, `y` is life expectancy, and the third variable is population:

```
fig, ax = plt.subplots()
ax.scatter(df['gdppc'], df['life'],     # x,y variables
            s=df['pop']/10**6,          # size of bubbles
            alpha=0.5)
ax.set_title('Life expectancy vs. GDP per capita', loc='left', fontsize=14)
ax.set_xlabel('GDP Per Capita')
ax.set_ylabel('Life Expectancy')
ax.text(58, 66, 'Bubble size represents population', horizontalalignment='right')
```

The only odd thing is the `10**6` "scaling" on the second line. The bubble size is a little tricky to calibrate. Without the scaling, the bubbles are larger than the graph. We played around until they looked reasonable.

Styles

Ok, we lied, that wasn't the conclusion. But we think this is fun, and it's optional in any case.

Matplotlib has a lot of basic settings for graphs. If we find some we like, we can set them once and be done with it. Or we can use some of their preset combinations, which they call **styles**.

We'll start with one of the bar charts we produced with World Bank data:

```
fig, ax = plt.subplots()
df['gdp'].plot(ax=ax, kind='barh', alpha=0.5)
ax.set_title('GDP', loc='left', fontsize=14)
ax.set_xlabel('Trillions of US Dollars')
ax.set_ylabel('')
```

Now recreate the same graph with this statement at the top:

```
plt.style.use('fivethirtyeight')
```

Once we execute this statement, it stays executed, but we'll change it back at the end.

Here's another one, for fans of the popular xkcd webcomic:

```
plt.xkcd()
fig, ax = plt.subplots()
df['gdp'].plot(ax=ax, kind='barh', alpha=0.5)
ax.set_title('GDP', loc='left', fontsize=14)
ax.set_xlabel('Trillions of US Dollars')
ax.set_ylabel('')
```

Note the wiggly lines, perfect for suggesting a hand-drawn graph.

Exercise. Try one of these styles: `ggplot` , `bmh` , `dark_background` , and `grayscale` . Which ones do you like? Why?

When we're done, we reset the style with these two lines in an code cell:

```
mpl.rcParams.update(mpl.rcParamsDefault)
%matplotlib inline
```

Review

Consider the data from Randal Olson's blog post:

```
import pandas as pd
data = {'Food': ['French Fries', 'Potato Chips', 'Bacon', 'Pizza', 'Chili Dog'],
        'Calories per 100g':  [607, 542, 533, 296, 260]}
cals = pd.DataFrame(data)
```

The dataframe `cals` contains the calories in 100 grams of several different foods.

Exercise. We'll create and modify visualizations of this data:

- Set `'Food'` as the index of `cals` .
- Create a bar chart with `cals` using figure and axis objects.
- Add a title.
- Change the color of the bars. What color do you prefer?
- Add the argument `alpha=0.5` . What does it do?
- Change your chart to a horizontal bar chart. Which do you prefer?
- *Challenging.* Eliminate the legend.
- *Challenging.* Skim the top of Olson's blog post. What do you see that you'd like to imitate?

Resources

A mercifully short markdown cheatsheet.

We haven't found many non-technical resources on Matplotlib we like, but these are pretty good:

- One of the best is Matplotlib's gallery of examples. It's a good starting point for learning new things. Find an example you like, download the code, and adapt it to your needs.

We also like the Pandas summary of dataframe methods.

- The documentation of Pandas plot methods is also pretty good.
- The SciPy lectures are good overall. The Matplotlib section focusses on `plot(x,y)`, which wouldn't be our choice, but the content is very good.
- Randal Olson has lots of good examples on his blog.

If you find others you like, let us know.