Data Bootcamp: Class #3

Revised: December 1, 2014

1 Course overview

- Objective: Learn enough about Python to do useful things with data.
- Target audience: Programming newbies. Anyone can do this with a little persistence and the help of friends.
- Team: Dave Backus, Glenn Okun, Sarah Beckett-Hile, and a rotating group of ninjas.
- Course resources:
  - Bootcamp Group: https://groups.google.com/forum/#!forum/nyu_data_bootcamp. Post comments and questions here.
  - GitHub repository: https://github.com/DaveBackus/Data_Bootcamp. All the docs and programs are here. This document is in the Notes folder; the pdf file comes with links. Programs are in the Code folder under Python.
  - This document: The online version comes with links.

2 Overall plan

- First class: Python basics, examples.
- Second class: graphics.
- This class: control flow, functions, data management.
- **We may run a more formal course in August.**

3 Today's plan

- Control flow: if statements, loops.
- Functions.
- Data management: the Pandas package.

4 Control flow language

Python has some basic features, or "control flow language," that are common to most programming languages. The most common are conditionals (do different things if a statement is true or false) and loops (do something a bunch of times). Each has a number of variations.

We'll use the term "syntax" to refer to the structure of the language, the set of rules that govern how we use commands. You might think of this as analogous to the rules of using English or some other language. The difference is that computer languages are much less complex.

- References
  - Python tutorial: https://docs.python.org/3/tutorial/controlflow.html. This is very good, we'll steal from it liberally.
  - SciPy Lectures: https://scipy-lectures.github.io/. This is written for a different audience (people in the natural sciences) but it's very good. See especially Section 1.2.
  - Wikpedia: http://en.wikipedia.org/wiki/Control_flow. A general overview of the concepts and their origins.
- Conditionals: `if`, `else`, `elif`. These statements allow you to do different things depending on the result of some condition. A really simple example prints the square of a number *if the number is greater than six* (that's the condition):

```
x = 7       # we can change this later and see what happens
if x > 6:
    square = x**2
    print(square)

print('Done!')
```

  The key elements of the code are:
  - The initial `if` statement ends with a colon. That's standard Python syntax, we'll see it again.
  - The following statements are indented four spaces. Spyder will do this automatically for you.
  - The end of the "code block" is signalled by the end of the indentation.
  - The space before the next line — `print('Done!')` — isn't necessary, but it's common practice because it makes the code easier to read.

  The starting point here is the comparison `x>6`. Comparisons take on the values `True` and `False`. In this case, if you type `print(x>6)` you'll get the answer: `True`. The `if` statement then directs the program to do different things in each case. The allowable comparisons are `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to).

  This example does something if the condition `x>6` is true, nothing if it's false. What if we want to do something else if the condition is false? Then we add an `else` statement:

```
if x > 6:
    square = x**2
    print('x**2 =', square)
    print(x>6)
else:
    print('x is not > 6 ( x =', x, ')')
    print(x>6)
```

We've added print statements for the condition `x>6` to show what they look like.

There are lots more variations, but that should give you the idea.

- **Exercise.** Start with the assignments

```
name1 = 'Dave'
name2 = 'Glenn'
```

(The names on the right can be anything, but let's start with these.) Write a program using `if` and `else` that prints out the name that comes first in alphabetical order.

- `for` loops. There are lots of times you want to do the same thing many times, either on one object or to many similar objects. An example of the former is to find an answer to progressively higher degrees of accuracy. We repeat as many time as we need to get a desired degree of accuracy. An example of the latter is to print out a list of names, one at a time. Both situations come up a lot.

  Here's an example: compute and print the squares of whole numbers (integers) up to ten. We can do that with a `for` loop:

```
for number in range(11):
    square = number**2
    print(square)
```

The variable name `number` is arbitrary, we can use any name we like. The expression `range(n)` sets up a sequence of integers of length `n`. It's a built-in Python tool. With standard Python logic, it goes from zero to `n-1`. The command `for` tells the program to do the commands that follow for all the values of `number` specified by `range(11)`; that is, for integer values between zero and ten: `0`, `1`, `2`, ... `10`. The print statement reports the result.

The syntax is similar to conditionals:

– The initial `for` statement ends with a colon.

– The following statements are indented four spaces.

– The end is signalled by the end of the indentation.

– A space after the last line makes the code easier to read.

Here's another example, this one adapted from the Python tutorial:

```
a, b = 0, 1
for it in range(10):
    a, b = b, a+b
    print('At iteration', it, 'b =', b)
```

This generates Fibonacci numbers (programmers love Fibonacci numbers): starting with zero and one, each subsequent number is the sum of the previous two. The print statement reports the iteration number `it` along with the next Fibonacci number `b` in the sequence.

Here's a more complex example. One of the features of Fibonacci numbers is that the ratio of successive numbers converges. But what does it converge to? We can compute the limit by repeating the operation and stopping when we think we're close enough. One measure of close enough is when the difference between successive ratios is smaller than some very small number. Here's an example:

```
a, b = 0, 1
ratio = a/b
maxit = 20
small_num = 1e-4
for it in range(maxit):
    a, b = b, a+b
    new_ratio = a/b
    print('At iteration', it, 'ratio =', new_ratio)
    if abs(new_ratio-ratio) < small_num:
        break       # exit loop
    else:
        ratio = new_ratio
```

Note:

- Additional indentation of interior code blocks (the `if` and `else` statements inside the `for` loop).
- The `break` command. This is new, it tells the program to exit the loop and go on to whatever is next in the program.

- **Exercise.** Use a `for` loop to compute the mean of the elements of the list: `x = [5, 7, 3, 6]`.
- **Exercise.** Write a program that computes the sum $1 + 2 + 3 + \cdots + 10$. Then modify it to compute the sum up to any positive integer n.
- `while` loops. We don't use them much, but the idea is to build the exit condition into the loop. Here's another example, taken from the Python tutorial, that generates the Fibonacci numbers up to 100:

```
a, b = 0, 1
while b < 100:
    print('b =', b)
    a, b = b, a+b
```

[Mini-exercise: what happens if we reverse the order of the two lines in the code block?]

Here's another example, a variant of an earlier one. Execution is the same, it's just a different way to write the code.

```
a, b = 0, 1
ratio = a/b
maxit = 20
small_num = 1e-4
error = 20
while error > small_num:
    a, b = b, a+b
    new_ratio = a/b
    print('At iteration', it, 'ratio =', new_ratio)
    error = abs(new_ratio-ratio)
    ratio = new_ratio
```

- `for` loops over lists and strings. We can also loop over lists and strings. In each case we run through their elements one at a time. Here's an example for a list:

```
fruit = ['apples', 'bananas', 'cherries']
for item in fruit:
    print(item)
```

- **Exercise.** This goes a little beyond what we've covered, but what do you think this program does? What features are new to you?

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

You should describe what every line does as well as the overall result. Extra credit: How would you change the program to print out the consonants? (This one is adapted from SciPy lecture 1.2.)

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

(a) Write a program that tells us the `type` of each element of `stuff`. (If you don't recall what the function `type` does, ask someone. Or type it in the Object inspector and read the documentation.)

(b) Write a program that prints out the elements of `stuff`.

(c) Write a program that goes through the elements of `stuff` and prints out only the elements that are strings; that is, the function `type` returns the value `str`.

- List comprehensions. That's a mouthful of jargon, but the idea is that we can use implicit loops with lists. (This is another thing that doesn't work in Python 2, so make sure you

have Python 3 installed.) Consider, for example, the loop above that prints out the elements of the list `fruit` one at a time. A list comprehension is a more compact syntax for the same thing:

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

5 Functions

Experts tell us that programmers never copy chunks of code. They write a function — once! — and call it as many times as they need. We wouldn't go that far, but it's pretty good advice.

[Skip for now]

6 Introduction to Pandas

Pandas is the leading data management package in Python. The name, often written in lower case letters, stands somehow for Python Data Analysis Library. Like Python, the name isn't designed with web searches in mind unless you're looking for pictures of animals, but you can search "python pandas" and get to the right place.

Pandas includes many of the tools commonly found in statistics programs for managing data. And to be clear: it's about organizing and managing data, not analysis in the sense of statistical tools. We'll get those from other packages when we need them.

The basic tool in Pandas is the DataFrame, which you might think of as similar to a worksheet of data. Like a worksheet, it has rows and columns. In a DataFrame, the rows are observations and the columns are variables. The DataFrame also has room for what you might call metadata, namely the labels of the rows and columns. The column labels are the variable names, strings that describe the variable in question. The row labels can be counters (number the observations in order), dates, or even multiple things (the date and country, say). It's an incredibly flexible tool once you get used to it.

- References
  - 10 minutes to Pandas: http://pandas.pydata.org/pandas-docs/stable/10min.html. This is excellent, covers most of what we need to get started.
  - Pandas via Excel: http://pbpython.com/excel-pandas-comp.html. Good short introduction to Pandas relating basic functions to the same in Excel.
  - Wes McKinney, *Python for Data Analysis*: http://www.amazon.com/dp/1449319793/. This is encylopedic, but the guy who wrote pandas, but we didn't find it all that easy to use.
  - The Pandas documentaion: http://pandas.pydata.org/pandas-docs/stable/. This includes a number of tutorials, but we haven't worked through them. Let us know if you find one you like.

- Examples.

  Example 1 (538 income data). The easiest way to create a DataFrame is to use a Pandas input method that sets one up automatically. This example comes from the 538 blog's data on income of recent graduates by college major. We read it from the 538's GitHub repository with the commands

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

  The first three lines give us the url of the data (it's too long to fit on one line, so we split it in two then put them together). The last line uses Pandas' tool for reading csv files; the command begins with `pd.` because it's part of the Pandas package and we imported pandas `as pd`.

  The end result is the DataFrame `df538`. We can verify that with the command `type(df538)`, which returns the answer: `pandas.core.frame.DataFrame`.

  What else do we have? We can look at the first and last five lines of data with the commands `df538.head()` and `df538.tail()`, but the formatting leaves something to be desired. The easiest way to get a list of variables is from the column labels: `df538.columns`. Sarah suggests this version, which gives us the output as a nicely formatted list, one item per row.

  Similarly, we get the observation labels from `df538.index` or, using the same trick, `df538.index.tolist()`. This is nicely formatted, too, but long: we have 173 observations (college majors) labelled (in the usual Python way) 0 to 172.

  Example 2 (make our own). We can also make our own DataFrame. Here's some code that does this from lists:

```
codes     = ['USA', 'FRA', 'JPN', 'CHN', 'IND', 'BRA', 'MEX']
countries = ['United States', 'France', 'Japan', 'China', 'India',
             'Brazil', 'Mexico']
gdppc = [53.1, 36.9, 36.3, 11.9, 5.4, 15.0, 16.5]  # thousands
gdp   = [16.8, 2.5, 4.7, 16.1, 6.8, 3.0, 2.1]  # trillions
df = pd.DataFrame([gdp, gdppc, countries]).T
```

  We saw most of this data last time. What we're doing here is adding a variable and converting it to a DataFrame. The `.T` transposes the data: for reasons we don't understand, it puts the lists into rows rather than columns. The `.T` reverses this.

  Our DataFrame `df` now has the data, but no row or column labels. We add them with

```
df.columns = ['gdp', 'gdppc', 'country']
df.index = codes
```

- **Exercise.** Take the data

```
x = [1, 2, 5, 7, 10]
y = [10, 7, 5, 2, 1]
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Put `x` and `y` into a DataFrame. Add variable names `x` and `y` and observation labels `days`.

- Selecting data. There are lots of ways of accessing subsets of the DataFrame, but here are some of the simplest. We can do more some other time.
  - Selecting observations. We can slice them as we did with strings and lists. Remember that numbering starts with zero. To get the first three observations of our GDP DataFrame `df` we type: `df[0:3]`. (Ask yourself: why 0:3?)
  - Selecting variables. This is cooler. We can refer to the variable `x` above equivalently as `df.x` or `df[x]`.

    We can also select subsets of data with a list of variable names. Here's an example for our GDP data:

    ```
    keep = ['gdp', 'country']
    other = df[keep]
    print(other)
    ```

- Constructing new variables. We can use similar syntax to construct new variables from old ones. For example, population is the ratio of GDP to GDP per capita. We can construct it from either of these lines of code:

```
df.pop = df.gdp/df.gdppc
df['pop'] = df['gdp']/df['gdppc']
```

7 More Pandas

When we have time, some things we could cover:

- Dates.
- Time aggregation (converting, say, monthly data to annual).
- Merging DataFrames.
- Hierarchical indexes.

Today's code: control

Attached. Download this pdf file, open in Adobe Acrobat or the equivalent, and click on the pushpins: 

Also here:

```
"""
For Class #3 of an informal mini-course at NYU Stern, Fall 2014.

Topics:  control flow, functions

Repository of materials (including this file):
* https://github.com/DaveBackus/Data_Bootcamp

Prepared by Dave Backus, Sarah Beckett-Hile, and Glenn Okun
Created with Python 3.4
"""

"""
Reminders
"""
x, y, z = 2*3, 2**3, 2/3    # yes, three at once!
a, b = 'some', 'thing'
c = a + b

numbers = [x, y, z]
strings = [a, b, c]
both = numbers + strings
print(both)
print(both[:3] + both[3:])

[print(element) for element in both]
[print(c[item]) for item in range(9)]
#%%
"""
Conditional statements:  if, else, elif
"""
# if statement
x = 5       # we can change this later and see what happens
if x > 6:
    square = x**2
    print('x**2 =', square)

print('Done!')

#%%
# if and else statements
x = 25
if x > 6:
    square = x**2
    print('x**2 =', square)
    print(x>6)
else:
    print('x is not > 6 ( x =', x, ')')
    print(x>6)

print('Done!')
```

```
#%%
name1 = 'Dave'
name2 = 'Allan'

if name1 > name2:
    print(name2)
else:
    print(name1)

#%%
"""
Loops
"""
# simple loop
for number in range(11):
    square = number**2
    print(square)

#%%
# Fibonacci numbers
print('\nFibonacci numbers')    # \n skips to the next line
a, b = 0, 1
for it in range(20):
    a, b = b, a+b
#    print('At iteration', it, 'b =', b)
    print('Ratio', a/b)

#%%
sum = 0
for num in range(1,11):
    sum += num
    print(sum)

#%%
a, b = 0, 1
ratio = a/b
maxit = 20
small_num = 1e-4
for it in range(maxit):
    a, b = b, a+b
    new_ratio = a/b
    print('At iteration', it, 'ratio =', new_ratio)
    if abs(new_ratio-ratio) < small_num:
        break       # stop and exit loop break
    else:
        ratio = new_ratio

#%%
# while loop
a, b = 0, 1
ratio = a/b
maxit = 20
```

```
small_num = 1e-4
error = 20
while error > small_num:
    a, b = b, a+b
    new_ratio = a/b
    print('At iteration', it, 'ratio =', new_ratio)
    error = abs(new_ratio-ratio)
    ratio = new_ratio

#%%
a, b = 0, 1
while b < 100:
    print('b =', b)
    a, b = b, a+b

#%%
# looping over strings and lists
vowels = 'aeiouy'
word   = 'anything'
for letter in word:
    if letter not in vowels:
        print(letter)

#%%
# loop over list
fruit = ['apples', 'bananas', 'cherries']
for item in fruit:
    print(item)

#%%
fruit = ['apples', 'bananas', 'cherries']
together = ''       # empty string
for item in fruit:
    print(item)
    together = together + item

print(together)
#%%

# list comprehension
[print(item) for item in fruit]

print('Done!')

#%%
"""
Functions
"""
# some other time

def fibonacci(a, b):
    a, b = b, a+b
```

```
    return a, b

a, b = 0, 1
a, b = fibonacci(a, b)

print(a, b)
```

Today's code: Pandas

Attached. Download this pdf file, open in Adobe Acrobat or the equivalent, and click on the pushpins: 

Also here:

```
"""
For Class #3 of an informal mini-course at NYU Stern, Fall 2014.

Topics:  pandas data management

Repository of materials (including this file):
* https://github.com/DaveBackus/Data_Bootcamp

Prepared by Dave Backus, Sarah Beckett-Hile, and Glenn Okun
Created with Python 3.4
"""
import pandas as pd

"""
Examples of DataFrames
"""
# 538's income by college major
url1 = 'https://raw.githubusercontent.com/fivethirtyeight/'
url2 = 'data/master/college-majors/recent-grads.csv'
url = url1 + url2
df538 = pd.read_csv(url)

#%%
# check to see what we have
df538.head()        # list first five observations
df538.tail()        # last five
#%%
# look at column names (.tolist() isn't necessary but easier to read)
df538.columns
df538.columns.tolist()

df538.index.tolist()

#%%
# Fama-French equity return data
import pandas.io.data as web
```

```
ff = web.DataReader('F-F_Research_Data_Factors', 'famafrench')[0]
ff.columns = ['xsm', 'smb', 'hml', 'rf']

#%%
# constructing a DataFrame from scratch
codes     = ['USA', 'FRA', 'JPN', 'CHN', 'IND', 'BRA', 'MEX']
countries = ['United States', 'France', 'Japan', 'China', 'India',
             'Brazil', 'Mexico']
# Wikipedia, 2013 USD
gdppc = [53.1, 36.9, 36.3, 11.9, 5.4, 15.0, 16.5]  # thousands
gdp   = [16.8, 2.5, 4.7, 16.1, 6.8, 3.0, 2.1]  # trillions

df = pd.DataFrame([gdp, gdppc, countries]).T

#df.columns = ['GDP (trillions of USD)', 'GDP per capita (thousands of USD)',
#              'Country Code']
df.columns = ['gdp', 'gdppc', 'country']
df.index = codes

#%%
# Spencer's version
# Warning:  uses a dictionary, skip if you don't know what that is
data = {"GDP (trillions of USD)": gdp,
        "GDP per capita (thousands of USD)": gdppc,
        "Country": countries}
dfspencer = pd.DataFrame(data, index=codes)

#%%
# Exercise
x = [1, 2, 5, 7, 10]
y = [10, 7, 5, 2, 1]
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']


"""
Selecting variables and observations
"""
# check
print(df)

some = df[0:3]
print(some)

keep_obs = [0, 1] #['USA', 'BRA', 'JPN']
keep_var = ['gdp', 'country']
other = df[keep_obs]
print(other)

print(df[0:3])


#%%
```

```
"""
Constructing new variables
"""
df.pop = df.gdp/df.gdppc
#%%
df['pop'] = df['gdp']/df['gdppc']
print(df)
```