Introduction to Data Science in Python

Nicholas P. Ross

November 14, 2016

Abstract

Here are my (NPR's) notes on the "Introduction to Data Science in Python" Coursera course from the University of Michigan that I'm taking in November 2016. The URL for that course is `https://www.coursera.org/learn/python-data-analysis/home/welcome`. The URL for these notes is: `https://github.com/d80b2t/Research_Notes/tree/master/Python`

Contents

1 Week 1: Python Fudamentals

1.1 Introduction to Specialization

Kinda a preamble!
General Course Outline (4 modules)
1. General Python Basics
2. The *pandas* Toolkit
3. Advanced Querying and Manipulation in *pandas*
4. Basic Statistical Analysis with *numpy* and *scipy*, and project.

1.2 Syllabus

`https://www.coursera.org/learn/python-data-analysis/supplement/68grE/syllabus`.

If you're having problems, here are a couple of great places to go for help:

- 1. If the problem is with the Coursera platform such as verification on assignments, in video quiz problems, or the Jupyter Notebooks, please check out the Coursera Learner Support Forums.
- 2. If the problem deals with understanding the assignment or how to use the Jupyter Notebooks, please read our Jupyter Notebook FAQ page in the course resources.
- 3. If you have questions with the content of the course, or questions about programming in python or with the toolkits described, you can contact your peers and the course instructors in the discussion forums, or go to Stack Overflow.

1.3 Data Science

`http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram`

David Donoho, Professor of Statistics in Stanford., "50 Years of Data Science". 1. Data Exploration and Preparation.
2. Data Representation and Transformation.
3. Computing with Data.
4. Data Modeling.
5. Data Visualization and Presentation.
6. Science about Data Science.

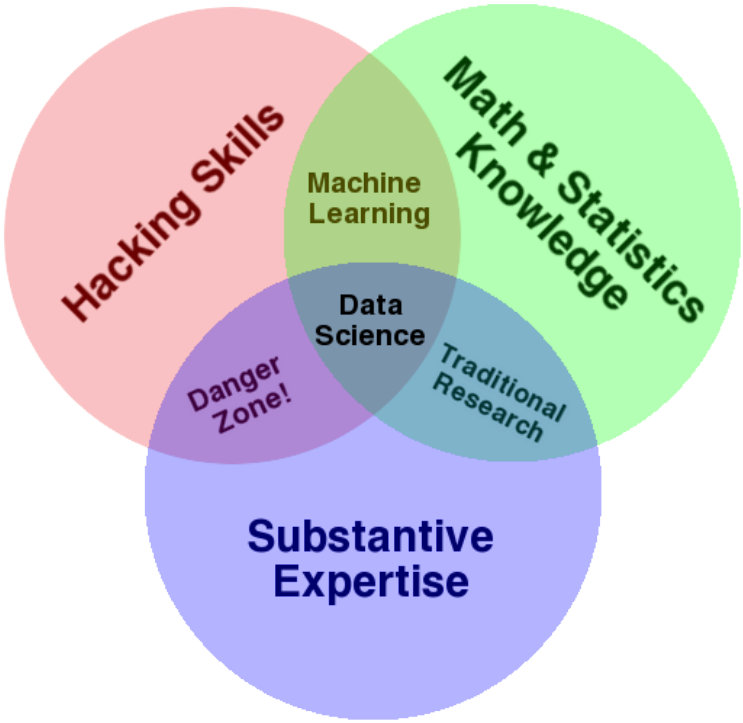

Figure 1: Drew Conway's Venn Diagram.

1.4 The Coursera Jupyter Notebook System

All pretty standard, straighforward.

1.5 Python Functions

Of course, Python has traditional software structures like functions. Here's an example, refactoring that previous code into a function. You'll see the def statement indicates that we're writing a function. Then each line that is part of the function needs to be indented with a tab character or a couple of spaces.

```
def add_numbers(x, y):
    return x + y

add_numbers(1, 2)
```

Okay, functions are great but they're a bit different than you might find in other languages and here are some of subtleties involved. First, since there's no typing, you don't have to set your return type. Second, you don't have to use a return statement at all actually. There's a special value called None that's returned. None is similar to null in Java and represents the absence of value. Third, in Python, you can have default values for parameters. Here's an example.

```
def add_numbers(x,y,z=None):
    if (z==None):
        return x+y
    else:
        return x+y+z

print(add_numbers(1, 2))
print(add_numbers(1, 2, 3))
```

In this example, we can rewrite the add numbers function to take three parameters, but we could set the last parameter to be None by default. This means that you can call add numbers with just two values or with three, and you don't have to rewrite the function signature to overload it.

```
def do_math(a, b, kind='add'):
  if (kind=='add'):
    return a+b
  else:
    return a-b
```

```
do_math(1, 2)
```

1.6 Python Types and Sequences

The absence of static typing in Python doesn't mean that there aren't types. The Python language has a built in function called type which will show you what type of given reference is. Some of the common types includes strings, the type is discussed. Integers and floating point variables. As we've seen you can have reference as to function as well as a function type also exist.

Typed objects have properties associated with them, and these properties can be data or functions. A lot of Python's built around different kinds of sequences or collection types. And there's three native kinds of collections that we're going to talk about, *tuples*, *lists*, and *dictionaries*.

1.6.1 Tuples

A tuple is a sequence of variables which itself is immutable. That means that a tuple has items in an ordering, but that it cannot be changed once created. We write tuples using parentheses, and we can mix types for the contents for the tuples. Here's a tuple which has four items. Two are numbers, and two are strings.

```
x = (1, 'a', 2, 'b')
type(x)
```

1.6.2 Lists

Lists are very similar, but they can be mutable, so you can change their length, number of elements, and the element values. A list is declared using the square brackets.

```
x = [1, 'a', 2, 'b']
type(x)
```

There are a couple of different ways to change the contents of a list. One is through the append function which allows you to append new items to the end of the list.

```
x.append(3.3)
print(x)
```

Both lists in tuple are iterable types, so you can write loops to go through every value they hold. The norm, if you want to look each item in the list is to use a `for` statement. This is similar to the for each loop in languages like Java and C# but note that there's no typing required.

```
for item in x:
print(item)
```

List and tuples can also be accessed as arrays might in other languages, by using the square brackets operator, which is called the indexing operator. The first item of the list starts at position zero and to get the length of the list, we use the built in lan function. There are some other common functions that you might expect like min and max which will find the minimum or maximum values in a given list or tuple.

```
i=0
while( i != len(x) ):
    print(x[i])
    i = i + 1
```

1.6.3 Other, really useful, `string` stuff

```
firstname = 'Christopher'
lastname = 'Brooks'
print(firstname + ' ' + lastname)
print(firstname*3)
print('Chris' in firstname)
```

`split` returns a list of all the words in a string, or a list split on a specific character.

```
firstname = 'Christopher Arthur Hansen Brooks'.split('
    ')[0] # [0] selects the first element of the list
lastname = 'Christopher Arthur Hansen Brooks'.split('
    ')[-1] # [-1] selects the last element of the list
print(firstname)
print(lastname)
```

1.6.4 Dictionaries

Dictionaries are similar to lists and tuples in that they hold a collection of items, but they're labeled collections which do not have an ordering. This means that for each value you insert into the dictionary, you must also give a key to get that value out. In other languages the structure is often called a map. And in Python we use curly braces to denote a dictionary. Here is an example where we might link names to email addresses. You can see that we indicate each item of the dictionary when creating it using a pair of values separated by colons. That you can retrieve a value for a given label using the indexing operator.

```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill
    Gates': 'billg@microsoft.com'}
x['Christopher Brooks'] # Retrieve a value by using the
    indexing operator

x['Kevyn Collins-Thompson'] = None
x['Kevyn Collins-Thompson']

## Iterate over all of the keys:
for name in x:
     print(x[name])

brooksch@umich.edu
None
billg@microsoft.com

## Iterate over all of the values:
for email in x.values():
    print(email)

## Iterate over all of the items in the list:

## You can unpack a sequence into different variables:
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

This last example is a little bit different, and it's an example of something called unpacking. In Python you can have sequence, that's a list or a tuple of values, and you can unpack those items into different variables through assignment in one statement.

1.7 Python More on Strings

In Python 3 strings are Unicode based, which led to the 256 characters in ASCII. But the world doesn't just run on Latin characters and there's a need to support non-English languages as well as characters which are not commonly used in words, but are commonly used elsewhere like mathematical operators. The Unicode Transformation Format, or UTF, is an attempt to solve this. It can be used to represent over a million different characters. This includes not only human languages like you might expect, but symbols like emojis too. Python 3 uses Unicode by default so there is no problem in dealing with international character sets.

```
sales_record = {
  'price': 3.24,
  'num_items': 4,
  'person': 'Chris'}

sales_statement = '{} bought {} item(s) at a price of {} each
    for a total of {}'

print(sales_statement.format(sales_record['person'],
                             sales_record['num_items'],
                             sales_record['price'],
                             sales_record['num_items']*sales_record['price']))
```

1.8 Python Demonstration: Reading and Writing CSV files

```
import csv
% precision 2
with open('mpg.csv') as csvfile:
    mpg = list(csv.DictReader(csvfile))

 mpg[:3] # The first three dictionaries in our list.
```

`csv.Dictreader` has read in each row of our csv file as a dictionary. `len` shows that our list is comprised of 234 dictionaries. `keys` gives us the column names of our csv:

```
mpg[0].keys()
```

This is how to find the average cty fuel economy across all cars. All values in the dictionaries are strings, so we need to convert to float.

```
sum(float(d['cty']) for d in mpg) / len(mpg)
## Wondering if 'd' here is some universal shorthand for the
    dict...??
```

Here's a more complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group.

```
cylinders = set(d['cyl'] for d in mpg)
cylinders

CtyMpgByCyl = []

for c in cylinders: # iterate over all the cylinder levels
    summpg = 0
    cyltypecount = 0
    for d in mpg: # iterate over all dictionaries
        if d['cyl'] == c: # if the cylinder level type matches,
            summpg += float(d['cty']) # add the cty mpg
            cyltypecount += 1 # increment the count

    CtyMpgByCyl.append((c, summpg / cyltypecount)) # append
        the tuple ('cylinder', 'avg mpg')

CtyMpgByCyl.sort(key=lambda x: x[0])
CtyMpgByCyl
```

1.9 Python Dates and Times

```
import datetime as dt
import time as tm

# time returns the current time in seconds since the Epoch.
    (January 1st, 1970)
tm.time()

# Convert the timestamp to datetime.
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow

dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute,
    dtnow.second
# get year, month, day, etc.from a datetime
delta = dt.timedelta(days = 100) # create a timedelta of 100
    days
delta

today = dt.date.today()

today - delta # the date 100 days ago

datetime.date(2016, 8, 7)

today > today-delta # compare dates
```