Introduction to Data Science in Python

Nicholas P. Ross

November 15, 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.

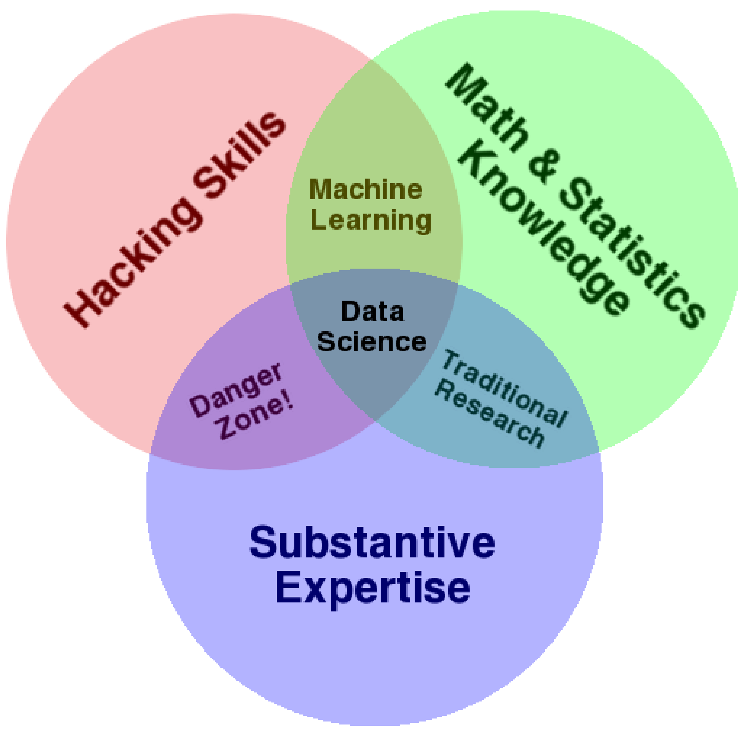

Figure 1: Drew Conway's Venn Diagram.

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.

```python
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.

```python
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.

```python
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

# get year, month, day, etc.from a datetime
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute,
    dtnow.second

# create a timedelta of 100 days
delta = dt.timedelta(days = 100)
delta

today = dt.date.today()

today - delta # the date 100 days ago

datetime.date(2016, 8, 7)

today > today-delta # compare dates
```

1.10 Advanced Python Objects, map()

Tiny, wee intro to OOP.
You can define a class using a class keyword, and ending with a colon. Anything indented below this, is within the scope of the class. An example of a class in python:

```
class Person:
    department = 'School of Information' # a class variable

    def set_name(self, new_name):         # a method
        self.name = new_name
    def set_location(self, new_location):
        self.location = new_location
```

Classes in Python are generally named using camel case, which means the first character of each word is capitalized.

You don't declare variables within the object, you just start using them. Class variables can also be declared. These are just variables which are shared across all instances. So in this example, we're saying that the default for all people is at the school of information.

To define a *method*, you just write it as you would have a function. The one change, is that to have access to the instance which a method is being invoked upon, you must include `self`, in the method signature. Similarly, if you want to refer to instance variables set on the object, you pre-pen them with the word `self`, with a full stop. In this definition of a person, for instance, we have written two methods, `set_name` and `set_location`. And both change instance bound variables, called name and location respectively.

Couple of key points on Python objects. First, objects in Python do not have private or protected member. If you instantiate an object, you have full access to any of the methods or attributes of that object. Second, there's no need for an explicit constructor when creating objects in Python. You can add a constructor if you want to by declaring the double underscore init double underscore method.

`map()`.
So, Functional programming is a programming paradigm in which you explicitly declare all parameters which could change through execution of a given function. Thus functional programming is referred to as being "side-effect free", because there is a software contract that describes what can

actually change by calling a function. Now, Python isn't a functional programming language in the pure sense. Since you can have many side effects of functions, and certainly you don't have to pass in the parameters of everything that you're interested in changing.

But functional programming causes one to think more heavily while chaining operations together. And this really is a sort of underlying theme in much of data science and date cleaning in particular. So, functional programming methods are often used in Python, and it's not uncommon to see a parameter for a function, be a function itself. The map built-in function is one example of a functional programming feature of Python, that I think ties together a number of aspects of the language. The map function signature looks like this. The first parameters of function that you want executed, and the second parameter, and every following parameter, is something which can be iterated upon.

```
store1 = [10.00, 11.00, 12.34, 2.34]
store2 = [9.00, 11.10, 12.34, 2.01]
cheapest = map(min, store1, store2)
cheapest

<map at 0x7fd424086eb8>
```

But when we go to print out the map, we see that we get an odd reference value instead of a list of items that we're expecting. This is called lazy evaluation. In Python, the map function returns to you a mapped object. Maps are iterable, just like lists and tuples, so we can use a for loop to look at all of the values in the map.

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

then gives you understandable O/P.

Here is a list of faculty teaching this MOOC. Can you write a function and apply it using map() to get a list of all faculty titles and last names (e.g. ['Dr. Brooks', 'Dr. Collins-Thompson', ...])??

```
people = ['Dr. Christopher Brooks', 'Dr. Kevyn
    Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel
    Romero']

def split_title_and_name(person):
    return #Your answer here

list(map(#Your answer here))
```

with an answer looking something like:

```
people = ['Dr. Christopher Brooks', 'Dr. Kevyn
    Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel
    Romero']

def split_title_and_name(person):
    title = person.split()[0]
    lastname = person.split()[-1]
    return '{} {}'.format(title, lastname)

list(map(split_title_and_name, people))
```

giving:

```
['Dr. Brooks', 'Dr. Collins-Thompson', 'Dr. Vydiswaran', 'Dr.
    Romero']
```

1.11 Advanced Python Lambda and List Comprehensions

Lambda's are Python's way of creating anonymous functions. These are the same as other functions, but they have no name. The intent is that they're simple or short lived and it's easier just to write out the function in one line instead of going to the trouble of creating a named function.

Convert this function in a lambda:

```
people = ['Dr. Christopher Brooks', 'Dr. Kevyn
    Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel
    Romero']

def split_title_and_name(person):
    return person.split()[0] + ' ' + person.split()[-1]

    #option 1
    for person in people:
        print(split_title_and_name(person) == (lambda
            person:???))

        #option 2
        #list(map(split_title_and_name, people)) ==
            list(map(???))
```

Something like this works well...

```
people = ['Dr. Christopher Brooks', 'Dr. Kevyn
    Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel
    Romero']

def split_title_and_name(person):
    return person.split()[0] + ' ' + person.split()[-1]

    #option 1
    for person in people:
        print(split_title_and_name(person) == (lambda x:
            x.split()[0] + ' ' + x.split()[-1])(person))

    #option 2
    list(map(split_title_and_name, people)) == list(map(lambda
        person: person.split()[0] + ' ' + person.split()[-1],
        people))
```

Noting the double "==" signs just to check the logic (vs. actually printing something out).

List Comprehensions.

```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

vs.

```
my_list = [number for number in range(0,1000) if number % 2 ==
    0]
my_list
```

Exercise: Convert function into a lise comprehension:

```
def times_tables():
    lst = []
    for i in range(10):
        for j in range (10):
            lst.append(i*j)
    return lst

times_tables() == [???]
```

```
def times_tables():
    lst = []
    for i in range(10):
        for j in range (10):
            lst.append(i*j)
    return lst

times_tables() == [j*i for i in range(10) for j in range(10)]
```

Noting the double "==" signs just to check the logic (vs. actually printing something out).

1.12 Advanced Python Demonstration: The Numerical Python Library (Numpy)

```
## Difference between
np.array([1, 2, 3] * 3)
np.repeat([1, 2, 3], 3)
```

```
a = np.array([-4, -2, 1, 3, 5])
a.sum()
a.max()
a.min()
a.mean()
a.std
## argmax and argmin return the index of the maximum and
    minimum values in the array
a.argmax()
a.argmin()
```

```
test = np.random.randint(0, 10, (4,3))
test
array(
[[8, 5, 3],
 [6, 0, 8],
 [3, 0, 0],
 [1, 8, 0]])

 test[0]
 array([8, 5, 3])

 test[0][0]
 8

 test[0,0]
 8

 test[1]
 array([6, 0, 8])

 test[0][1]
 5

 test[0, 1]
 5
```

```
for i, row in enumerate(test):
    print('row', i, 'is', row)

row 0 is [8 5 3]
row 1 is [6 0 8]
row 2 is [3 0 0]
row 3 is [1 8 0]

# Use `zip` to iterate over multiple iterables.

test2 = test**2
for i, j in zip(test, test2):
    print(i,'+',j,'=',i+j)

[8 5 3] + [64 25 9] = [72 30 12]
[6 0 8] + [36 0 64] = [42 0 72]
[3 0 0] + [9 0 0] = [12 0 0]
[1 8 0] + [ 1 64 0] = [ 2 72 0]
```

Week One quiz notes: Python is an example of an Interpreted language.
In Python, strings are considered INmutable.
When you create a lambda, what type is returned? A function.

2 Week Two: Basic Data Processing with Pandas

2.1 Introduction

GoTo: http://stackoverflow.com/.
Good RSS: Planet Python.
Podcast: ttp://dataskeptic.com/http://dataskeptic.com/

2.2 The Series Data Structure

`pd.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)`

If we create a list of strings and we have one element, a None type, pandas inserts it as a None and uses the type object for the underlying array:

```
animals = ['Tiger', 'Bear', None]
print(type(pd.Series(animals)))
pd.Series(animals)

<class 'pandas.core.series.Series'>

0    Tiger
1     Bear
2     None
dtype: object
```

If we create a list of numbers, integers or floats, and put in the None type, pandas automatically converts this to a special floating point value designated as NAN, which stands for not a number:

```
numbers = [1, 2, None]
print(type(pd.Series(numbers)))
pd.Series(numbers)

<class 'pandas.core.series.Series'>

0    1.0
1    2.0
2    NaN
dtype: float64
```

N.B the NaN; **NAN is not none and when we try the equality test, it's false.**

2.3 Querying a Series

2.4 The DataFrame Data Structure

2.5 DataFrame Indexing and Loading

2.6 Querying a DataFrame

2.7 Indexing Dataframes

2.8 Missing Values

2.9 Discussion Prompt: The Ethics of Using Hacked Data

3 Week Three

3.1 Merging Dataframes

3.2 Pandas Idioms

3.3 Group by

3.4 Scales

3.5 Pivot Tables

3.6 Date Functionality

4 Week Four: Statistical Analysis in Python and Project

4.1 Introduction

4.2 Distributions

4.3 More Distributions

4.4 Hypothesis Testing in Python

5 References and Bibliography