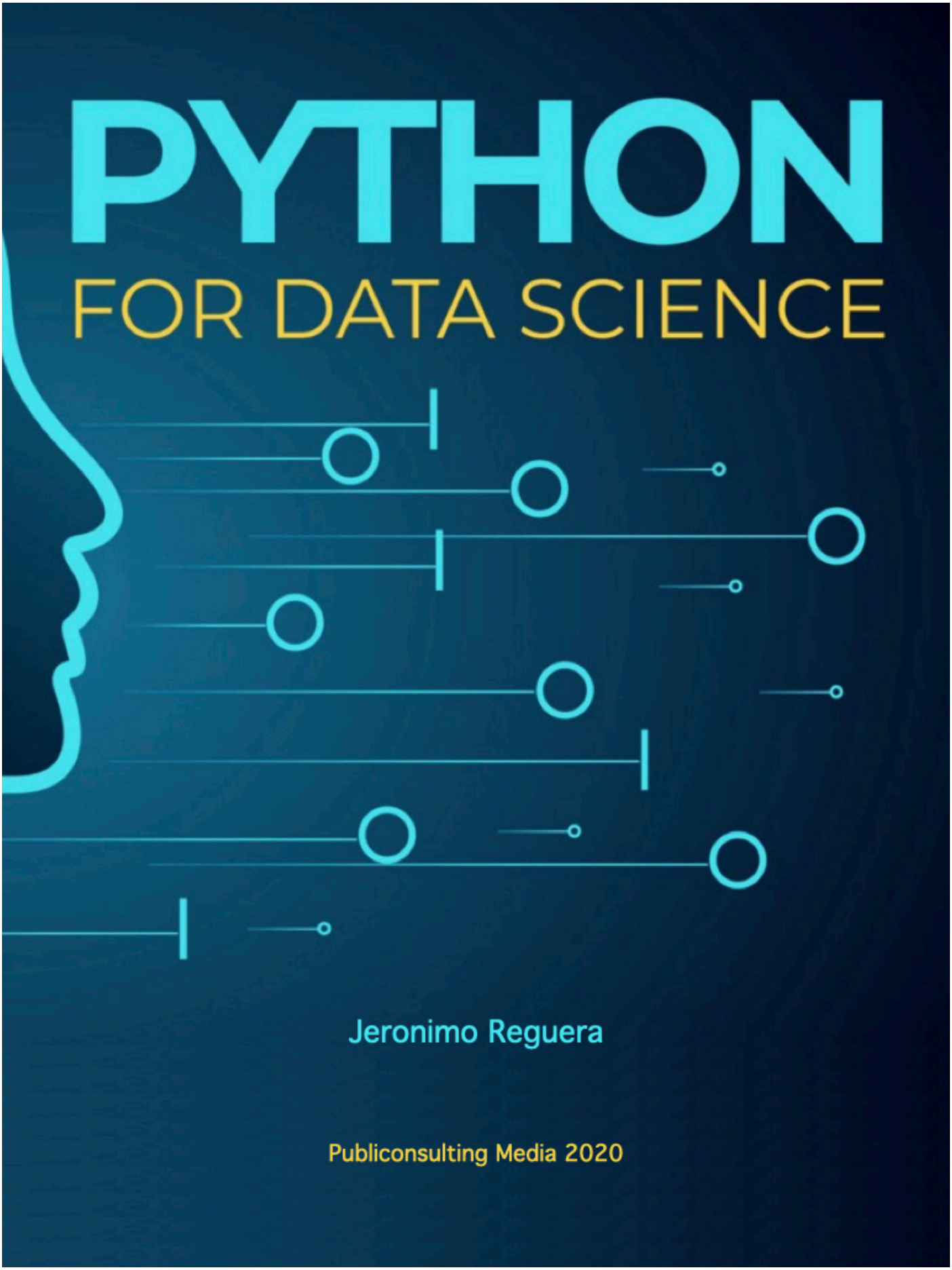
Python for Data Science

PYTHON
FOR DATA SCIENCE



Jeronimo Reguera

Publiconsulting Media 2020

Python for Data Science

Python for Data Science

JERONIMO REGUERA

PUBLICONSULTING MEDIA

Contents

Introduction

This is just an example of a Pressbooks book using Jupyter Notebook to add interactivity to the many great functionalities already available at Pressbooks platform, such as reader annotations (Hypothesis) and interactive content (H5P).

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

It is very easy to include Jupyter notebooks in your open textbooks using a plugin by Andrew Challis.

In this example book, lectures and assignments are provided as interactive Jupyter notebooks. Readers can then study through a lecture notebook and then read the assignment notebook and access it through GitHub to open and run the Jupyter notebook.

Installation

1. Install WP Pusher as a plugin in your Pressbooks site by downloading and installing the package from a zip file: go to your Pressbooks dashboard, Network Administrator, and select Plugins -> Add Plugins -> Upload Plugin. Upload your WP Pusher zip file, install it and activate it
2. Install the **nbconvert plugin** in your Pressbooks site: from the WP Pusher settings, add the nbconvert plugin by typing this uri into the git plugin installer: ghandic/nbconvert. Go to Network Administrator -> Plugins -> WP Pusher -> Install plugin -> [Repository host: select GitHub] -> [Plugin repository: type ghandic/nbconvert] -> Install plugin
3. Activate the nbconvert plugin for your network
4. Customize your book css as needed, since nbconvert.css from the plugin can require some adaptation (in your book go to Appearance -> Custom Styles -> Your Web Styles). My custom styles file is available here.

How it works

Simply add a shortcode and a url to the notebook file into your page editor on Pressbooks, between square brackets:

nbconvert url=”https://github.com/Jero2760/nbconvert/blob/master/example1.ipynb”

Lecture 1

Check it out on github
Last updated: 09/05/2020 08:38:51

In [ ]:

```
%matplotlib inline
import matplotlib
import seaborn as sns
matplotlib.rcParams['savefig.dpi'] = 144
```

In [2]:

```
import expectexception
```

Object Oriented Programming¶

You have probably already heard of **Python objects** or **objects** in general. You may have even heard the term **method** mentioned as well. What do these terms mean?

For now we can think of an object as anything we can store in a variable. We can have objects with different `type`. We might also call an object's `type` its **class**. We'll come back to class later.

In [3]:

```
x = 42
print('%d is an object of %s' % (x, type(x)))

x = 'Hello world!'
print('%s is an object of %s' % (x, type(x)))

x = {'name': 'Dylan', 'age': 26}
print('%s is an object of %s' % (x, type(x)))
```

```
42 is an object of <class 'int'>
Hello world! is an object of <class 'str'>
{'name': 'Dylan', 'age': 26} is an object of <class 'dict'>
```

We already know that integers, strings, and dictionaries behave differently. They have different properties and different capabilities. In the language of programming, we say they have different **attributes** and **methods**.

An object's attributes are its internal variables that are used to store information about the object.

In [10]:

```
# a complex number has real and imaginary parts
x = complex(5, 3)
print(x)
print(x.real)
print(x.imag)
```

```
(5+3j)
5.0
```

```
3.0
```

An object's methods are its internal functions that implement different capabilities.

In [11]:

```
x = 'Dylan'
print(x.lower())
print(x.upper())
```

```
dylan
DYLAN
```

We'll interact with an object's methods more often than its attributes. The attributes represent the *state* of an object. We usually prefer to mutate the state of an object via its methods, since the methods represent the actions one can take safely without breaking the object. Often the attributes of an object will be immutable.

In [12]:

```
%%expect_exception AttributeError

x = complex(5, 3)
x.real = 6
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-12-947651b88b0e> in <module>()
      1
      2 x = complex(5, 3)
----> 3 x.real = 6

AttributeError: readonly attribute
```

An example of a method that mutates an object is the `append` method of a `list`.

In [13]:

```
x = [35, 'example', 348.1]
x.append(True)
print(x)
```

```
[35, 'example', 348.1, True]
```

How do we know what the attributes and methods of an object are? We can use Python's `dir` function.

We can use `dir` on an object or on a class.

In [14]:

```
# dir on an object
x = 42
print(dir(x)[-6:]) # I've truncated the results for clarity

# dir on a class
print(dir(int)[-6:])
```

```
['denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
['denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

We can also look up documentation on the class. For example, here's Python's documentation on the built-in Python types. We'll use documentation more and more as we incorporate third-party libraries and tools into Python.

Classes¶

But this isn't the whole story. The methods and attributes of a `dict` don't tell us anything about key-value pairs or hashing. The full definition of an object is an object's class. We can define our own classes to create objects that carry out a variety of related tasks or represent information in a convenient way. Some examples we'll deal with later in the course are classes for making plots and graphs, classes for creating and analyzing tables of data, and classes for doing statistics and regression.

For now, let's implement a class called `Rational` for working with fractional numbers (e.g. 5/15). The first thing we'll need `Rational` to do is to be able to create a `Rational` object. We define how this should work with a special (hidden) method called `__init__`. We'll also define another special method called `__repr__` that tells Python how to print out the object.

In [20]:

```
class Rational(object):

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __repr__(self):
        return '%d/%d' % (self.numerator, self.denominator)
```

In [21]:

```
fraction = Rational(4, 3)
print(fraction)
```

```
4/3
```

You might have noticed that both of the methods took as a first argument the keyword `self`. The first argument to any method in a class is the instance of the class upon which the method is being called. Think of a class like a blueprint from which possibly many objects are built. The `self` argument is the mechanism Python uses so that the method can know which instance of the class it is being called upon. When the method is actually called, we can call it in two ways. Lets say we create a class `MyClass` with method `.do_it(self)`, if we instantiate an object from this class, we can call the method in two ways:

In [22]:

```
class MyClass(object):
    def __init__(self, num):
        self.num = num
    def do_it(self):
        print(self.num)
myclass = MyClass(2)
myclass.do_it()
MyClass.do_it(myclass)
```

```
2
2
```

In on way `myclass.do_it()` the `self` argument is understood because `myclass` is an instance of `MyClass`. This is the almost universal way to do call a method. The other possibility is `MyClass.do_it(myclass)` where we are passing in the object `myclass` as the `self` argument, this syntax is much less common.

Like all Python arguments, there is no need for `self` to be named `self`, we could also call it `this` or `apple` or `wizard`. However, the use of `self` is a very strong Python convention which is rarely broken. You should use this convention so that your code is understood by other people.

Lets get back to our `Rational` class. So far, we can make a `Rational` object and `print` it out, but it can't do much else. We might also want a `reduce` method that will divide the numerator and denominator by their greatest common divisor. We will therefore need to write a function that computes the greatest common divisor. We'll add these to our class definition.

In [29]:

```
class Rational(object):

    def __init__(self, numerator, denominator):
        self.numerator = numerator
```

```
        self.denominator = denominator

    def __repr__(self):
        return '%d/%d' % (self.numerator, self.denominator)

    def _gcd(self):
        smaller = min(self.numerator, self.denominator)
        small_divisors = {i for i in range(1, smaller + 1) if smaller % i == 0}
        larger = max(self.numerator, self.denominator)
        common_divisors = {i for i in small_divisors if larger % i == 0}
        return max(common_divisors)

    def reduce(self):
        gcd = self._gcd()
        self.numerator = self.numerator / gcd
        self.denominator = self.denominator / gcd
        return self
```

In [34]:

```
fraction = Rational(16, 32)
print('Fraction to reduce',fraction)
fraction.reduce()
print('Reduced Fraction',fraction)
```

```
Fraction to reduce 16/32
Reduced Fraction 1/2
```

We're gradually building up the functionality of our `Rational` class, but it has a huge problem: we can't do math with it!

In [35]:

```
%%expect_exception TypeError

print(4 * fraction)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-cc5282b78078> in <module>()
      1
----> 2 print(4 * fraction)

TypeError: unsupported operand type(s) for *: 'int' and 'Rational'
```

We have to tell Python how to implement mathematical operators (`+`, `-`, `*`, `/`) for our class.

In [36]:

```
print(dir(int))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
'__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

In [39]:

```
x = int(8)
x.__add__(9)
```

Out[39]:

```
17
```

If we look at `dir(int)` we see it has hidden methods like `__add__`, `__div__`, `__mul__`, `__sub__`, etc. Just like `__repr__` tells Python how to `print` our object, these hidden methods tell Python how to handle mathematical operators.

Let's add the methods implementing mathematical operations to our class definition. To perform addition or subtraction, we'll have to find a common denominator with the number we're adding. For simplicity, we'll only implement multiplication. We won't be able to add, subtract, or divide. Even implementing only multiplication will require quite a bit of logic.

In [41]:

```
class Rational(object):

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __repr__(self):
        return '%d/%d' % (self.numerator, self.denominator)

    def __mul__(self, number):
        if isinstance(number, int):
            return Rational(self.numerator * number, self.denominator)
        elif isinstance(number, Rational):
```

```
            return Rational(self.numerator * number.numerator, self.denominator *
number.denominator)
        else:
            raise TypeError('Expected number to be int or Rational. Got %s' % type(number))
    def _gcd(self):
        smaller = min(self.numerator, self.denominator)
        small_divisors = {i for i in range(1, smaller + 1) if smaller % i == 0}
        larger = max(self.numerator, self.denominator)
        common_divisors = {i for i in small_divisors if larger % i == 0}
        return max(common_divisors)

    def reduce(self):
        gcd = self._gcd()
        self.numerator = self.numerator / gcd
        self.denominator = self.denominator / gcd
        return self
```

In [44]:

```
print(Rational(4, 6) * 3)
print(Rational(5, 9) * Rational(2, 3))
```

```
12/6
10/27
```

In [45]:

```
%%expect_exception TypeError

# remember, no support for float
print(Rational(4, 6) * 2.3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-e585b376123d> in <module>()
      1
      2 # remember, no support for float
----> 3 print(Rational(4, 6) * 2.3)

<ipython-input-41-bfab24d3b796> in __mul__(self, number)
     14             return Rational(self.numerator * number.numerator, self.denominator *
number.denominator)
     15         else:
---> 16             raise TypeError('Expected number to be int or Rational. Got %s' %
type(number))
     17
     18     def _gcd(self):

TypeError: Expected number to be int or Rational. Got <class 'float'>
```

In [46]:

```
%%expect_exception TypeError

# also, no addition, subtraction, etc.
print(Rational(4, 6) + Rational(2, 3))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-3e2914851db1> in <module>()
      1
      2 # also, no addition, subtraction, etc.
----> 3 print(Rational(4, 6) + Rational(2, 3))

TypeError: unsupported operand type(s) for +: 'Rational' and 'Rational'
```

Defining classes can be a lot of work. We have to imagine all the ways we might want to use an object, and where we might run into trouble. This is also true of defining functions, but classes will typically handle many tasks while a function might only do one.

Private Methods in Python¶

You might have noticed we have used some methods which start with _ such as `_gcd`. This has a conventional meaning in Python which is formally implemented in other languages, the notion of a private function. Classes are used to encapsulate functionality and data while providing an interface to the outside world of other objects. Think of a program as a company, each worker has their own responsibilities and they know that other people the company perform certain tasks, but they don't necessary know how those people perform those tasks.

In order to make this possible, Classes have both public and private methods. Public methods are methods which are exposed to other objects or user interaction. Private methods are used internally to the object, often in a "helper" sense. In some languages this notion of public and private methods is enforced and the programmer will have to specify every method as either public or private. In Python every method is public, but to distinguish which methods we mean to be private, we add an underscore to the front of the method, hence `_gcd`. This is a note to someone using the class that this method should only be called inside the object and can be subject to change with new versions, whereas the public methods will hopefully not change their interface.

Another Python convention dealing with underscores are the so called `dunder` methods which have double underscores before and after the method names. There are a bunch of these in Python `__init__`, `__name__`, `__add__`, etc and they have special meaning. Note that they are generally considered private

methods as well except in special circumstances. In the case of methods like `__add__`, they are what allow the programmer to specify the + operation. Since these method have special meaning to Python they should only be used with care. Additionally, even though overloading things like the + operator might make sense to you as you program it, it can very confusing to someone reading your code as Python's dynamic type system usually does not allow determination of types until runtime, usually defining an `.add` method is much more clear.

When do we want Classes?¶

When we want to perform a set of related tasks, especially in repetition, we will usually want to define a new class. We will see that in most of the third-party libraries we will use, the major tools they introduce to Python are new classes. For example, later in the course we'll learn about the Pandas library, whose main feature is the `DataFrame` class.

In [47]:

```
import pandas as pd

df = pd.DataFrame({'a': [1, 2, 5], 'b': [True, False, True]})

print(type(df))
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

Out[47]:

	a	b
0	1	True
1	2	False
2	5	True

Here's the (abridged) beginning of the DataFrame class definition:

```
class DataFrame(NDFrame):

    def __init__(self, data=None, index=None, columns=None, dtype=None,
                 copy=False):
        if data is None:
            data = {}
        if dtype is not None:
            dtype = self._validate_dtype(dtype)
```

```
        if isinstance(data, DataFrame):
            data = data._data

        if isinstance(data, BlockManager):
            mgr = self._init_mgr(data, axes=dict(index=index, columns=columns),
                                 dtype=dtype, copy=copy)
        elif isinstance(data, dict):
            mgr = self._init_dict(data, index, columns, dtype=dtype)
        elif isinstance(data, ma.MaskedArray):
            import numpy.ma.mrecords as mrecords
            # masked recarray
            if isinstance(data, mrecords.MaskedRecords):
                mgr = _masked_rec_array_to_mgr(data, index, columns, dtype,
                                               copy)

            # a masked array
            else:
                mask = ma.getmaskarray(data)
                if mask.any():
                    data, fill_value = maybe_upcast(data, copy=True)
                    data[mask] = fill_value
                else:
                    data = data.copy()
                mgr = self._init_ndarray(data, index, columns, dtype=dtype,
                                         copy=copy)

        elif isinstance(data, (np.ndarray, Series, Index)):
            if data.dtype.names:
                data_columns = list(data.dtype.names)
                data = dict((k, data[k]) for k in data_columns)
                if columns is None:
                    columns = data_columns
                mgr = self._init_dict(data, index, columns, dtype=dtype)
            elif getattr(data, 'name', None) is not None:
                mgr = self._init_dict({data.name: data}, index, columns,
                                      dtype=dtype)
            else:
                mgr = self._init_ndarray(data, index, columns, dtype=dtype,
                                         copy=copy)
        elif isinstance(data, (list, types.GeneratorType)):
            if isinstance(data, types.GeneratorType):
                data = list(data)
            if len(data) > 0:
                if is_list_like(data[0]) and getattr(data[0], 'ndim', 1) == 1:
                    if is_named_tuple(data[0]) and columns is None:
                        columns = data[0]._fields
                    arrays, columns = _to_arrays(data, columns, dtype=dtype)
                    columns = _ensure_index(columns)

                    # set the index
```

```
                if index is None:
                    if isinstance(data[0], Series):
                        index = _get_names_from_index(data)
                    elif isinstance(data[0], Categorical):
                        index = _default_index(len(data[0]))
                    else:
                        index = _default_index(len(data))

                mgr = _arrays_to_mgr(arrays, columns, index, columns,
                                     dtype=dtype)
            else:
                mgr = self._init_ndarray(data, index, columns, dtype=dtype,
                                         copy=copy)
        else:
            mgr = self._init_dict({}, index, columns, dtype=dtype)
    elif isinstance(data, collections.Iterator):
        raise TypeError("data argument can't be an iterator")
    else:
        try:
            arr = np.array(data, dtype=dtype, copy=copy)
        except (ValueError, TypeError) as e:
            exc = TypeError('DataFrame constructor called with '
                            'incompatible data and dtype: %s' % e)
            raise_with_traceback(exc)

        if arr.ndim == 0 and index is not None and columns is not None:
            values = cast_scalar_to_array((len(index), len(columns)),
                                          data, dtype=dtype)
            mgr = self._init_ndarray(values, index, columns,
                                     dtype=values.dtype, copy=False)
        else:
            raise ValueError('DataFrame constructor not properly called!')

    NDFrame.__init__(self, mgr, fastpath=True)
```

That's a lot of code just for `__init__`!

Often we'll use the relationship between a new class and existing classes to *inherit* functionality, saving us from writing some code.

Inheritance¶

Often the classes we define in Python will build off of existing ideas in other classes. For example, our `Rational` class is a number, so it should behave like other numbers. We could write an implementation

of `Rational` that uses `float` arithmetic and simply converts between floating point and rational representations during input and output. This would save us complexity in implementing the arithmetic, but might complicate object creation and representation. Even if you never write a class, it's useful to understand the idea of inheritance and the relationship between classes.

Lets write a general class called `Rectangle`, it will have two attributes, a length and a width, as well as a few methods.

In [48]:

```
class Rectangle(object):
    def __init__(self, height, length):
        self.height = height
        self.length = length
    def area(self):
        return self.height * self.length
    def perimeter(self):
        return 2 * (self.height + self.length)
```

Now a square is also a rectangle, but its somewhat more restricted in that it has the same height as length, so we can subclass `Rectangle` and enforce this in code.

In [49]:

```
class Square(Rectangle):
    def __init__(self, length):
        super(Square, self).__init__(length, length)
```

In [50]:

```
s = Square(5)
s.area(), s.perimeter()
```

Out[50]:

```
(25, 20)
```

Sometimes (although not often) we want to actually check the type of a python object (what class it is from). There are two ways of doing this, lets first look at a few examples to get a sense of the difference.

In [51]:

```
type(s) == Square
```

Out[51]:

```
True
```

In [52]:

```
type(s) == Rectangle
```

Out[52]:

```
False
```

In [53]:

```
isinstance(s, Rectangle)
```

Out[53]:

```
True
```

As you might have noticed checking type quality only checks the exact class to which an object belongs, whereas `isinstance(c, Class)` checks if `c` is either a member of class `Class` or a member of a subclass of `Class`. Almost always `isinstance` is the proper way to check this, because if a class implements some sort of functionality, its subclasses usually implement the same functionality (they just might have some extra bonus functionality!).

Object Oriented Programming¶

Now that we understand objects and classes, let's return to the idea of *object oriented programming*. Object oriented programming (`OOP`) is a perspective that programs are essentially about the creation of objects and the interaction between them. In `OOP`, almost every piece of code either describes an object, an object's attributes, or an object's methods. Keeping this perspective in mind can help us understand what's happening in a program.

Questions:

An interactive or media element has been excluded from this version of the text. You can view it online here: https://www.publiconsulting.com/wordpress/pythonfords/?p=5

Assignment 1

Check it out on github
Last updated: 09/05/2020 08:38:51

In [ ]:

```
%matplotlib inline
import matplotlib
import seaborn as sns
matplotlib.rcParams['savefig.dpi'] = 144
```

PW Miniproject¶

Introduction¶

The objective of this miniproject is to exercise your ability to use basic Python data structures, define functions, and control program flow. We will be using these concepts to perform some fundamental data wrangling tasks such as joining data sets together, splitting data into groups, and aggregating data into summary statistics.
Please do not use `pandas` or `numpy` to answer these questions.

We will be working with medical data from the British NHS on prescription drugs. Since this is real data, it contains many ambiguities that we will need to confront in our analysis. This is commonplace in data science, and is one of the lessons you will learn in this miniproject.

Downloading the data¶

We first need to download the data we'll be using from Amazon S3:

In [ ]:

```
%%bash
mkdir pw-data
```

```
wget http://dataincubator-wqu.s3.amazonaws.com/pwdata/201701scripts_sample.json.gz -nc -P ./pw-data
wget http://dataincubator-wqu.s3.amazonaws.com/pwdata/practices.json.gz -nc -P ./pw-data
```

Loading the data¶

The first step of the project is to read in the data. We will discuss reading and writing various kinds of files later in the course, but the code below should get you started.

In [ ]:

```
import gzip
import simplejson as json
```

In [ ]:

```
with gzip.open('./pw-data/201701scripts_sample.json.gz', 'rb') as f:
    scripts = json.load(f)

with gzip.open('./pw-data/practices.json.gz', 'rb') as f:
    practices = json.load(f)
```

This data set comes from Britain's National Health Service. The `scripts` variable is a list of prescriptions issued by NHS doctors. Each prescription is represented by a dictionary with various data fields: `'practice'`, `'bnf_code'`, `'bnf_name'`, `'quantity'`, `'items'`, `'nic'`, and `'act_cost'`.

In [ ]:

```
scripts[:2]
```

A glossary of terms and FAQ is available from the NHS regarding the data. Below we supply a data dictionary briefly describing what these fields mean.

Data field	Description
'practice'	Code designating the medical practice issuing the prescription
'bnf_code'	British National Formulary drug code
'bnf_name'	British National Formulary drug name
'quantity'	Number of capsules/quantity of liquid/grams of powder prescribed
'items'	Number of refills (e.g. if 'quantity' is 30 capsules, 3 'items' means 3 bottles of 30 capsules)
'nic'	Net ingredient cost
'act_cost'	Total cost including containers, fees, and discounts

The practices variable is a list of member medical practices of the NHS. Each practice is represented by a dictionary containing identifying information for the medical practice. Most of the data fields are self-explanatory. Notice the values in the 'code' field of practices match the values in the 'practice' field of scripts.

In [ ]:

```
practices[:2]
```

In the following questions we will ask you to explore this data set. You may need to combine pieces of the data set together in order to answer some questions. Not every element of the data set will be used in answering the questions.

Question 1: summary_statistics¶

Our beneficiary data (scripts) contains quantitative data on the number of items dispensed ('items'), the total quantity of item dispensed ('quantity'), the net cost of the ingredients ('nic'), and the actual cost to the patient ('act_cost'). Whenever working with a new data set, it can be useful to calculate summary statistics to develop a feeling for the volume and character of the data. This makes it easier to spot trends and significant features during further stages of analysis.

Calculate the sum, mean, standard deviation, and quartile statistics for each of these quantities. Format your results for each quantity as a list: [sum, mean, standard deviation, 1st quartile, median, 3rd quartile]. We'll create a tuple with these lists for each quantity as a final result.

In [ ]:

```
def describe(key):
```

```
    total = 0
    avg = 0
    s = 0
    q25 = 0
    med = 0
    q75 = 0

    return (total, avg, s, q25, med, q75)
```

In [ ]:

```
summary = [('items', describe('items')),
           ('quantity', describe('quantity')),
           ('nic', describe('nic')),
           ('act_cost', describe('act_cost'))]
```

Question 2: most_common_item¶

Often we are not interested only in how the data is distributed in our entire data set, but within particular groups -- for example, how many items of each drug (i.e. `'bnf_name'`) were prescribed? Calculate the total items prescribed for each `'bnf_name'`. What is the most commonly prescribed `'bnf_name'` in our data?

To calculate this, we first need to split our data set into groups corresponding with the different values of `'bnf_name'`. Then we can sum the number of items dispensed within in each group. Finally we can find the largest sum.

We'll use `'bnf_name'` to construct our groups. You should have *5619* unique values for `'bnf_name'`.

In [ ]:

```
bnf_names = ...
assert(len(bnf_names) == 5619)
```

We want to construct "groups" identified by `'bnf_name'`, where each group is a collection of prescriptions (i.e. dictionaries from `scripts`). We'll construct a dictionary called `groups`, using `bnf_names` as the keys. We'll represent a group with a `list`, since we can easily append new members to the group. To split our `scripts` into groups by `'bnf_name'`, we should iterate over `scripts`, appending prescription dictionaries to each group as we encounter them.

In [ ]:

```
groups = {name: [] for name in bnf_names}
```

```
for script in scripts:
    # INSERT ...
```

Now that we've constructed our groups we should sum up `'items'` in each group and find the `'bnf_name'` with the largest sum. The result, `max_item`, should have the form `[(bnf_name, item total)]`, e.g. `[('Foobar', 2000)]`.

In [ ]:

```
max_item = [("", 0)]
```

Challenge: Write a function that constructs groups as we did above. The function should accept a list of dictionaries (e.g. `scripts` or `practices`) and a tuple of fields to `groupby` (e.g. `('bnf_name')` or `('bnf_name', 'post_code')`) and returns a dictionary of groups. The following questions will require you to aggregate data in groups, so this could be a useful function for the rest of the miniproject.

In [ ]:

```
def group_by_field(data, fields):
    groups = {}
    return groups
```

In [ ]:

```
groups = group_by_field(scripts, ('bnf_name',))
test_max_item = ...

assert test_max_item == max_item
```

Question 3: postal_totals¶

Our data set is broken up among different files. This is typical for tabular data to reduce redundancy. Each table typically contains data about a particular type of event, processes, or physical object. Data on prescriptions and medical practices are in separate files in our case. If we want to find the total items prescribed in each postal code, we will have to *join* our prescription data (`scripts`) to our clinic data (`practices`).

Find the total items prescribed in each postal code, representing the results as a list of tuples `(post code, total items prescribed)`. Sort your results ascending alphabetically by post code and take only results from the first 100 post codes. Only include post codes if there is at least one prescription from a practice in that post code.

NOTE: Some practices have multiple postal codes associated with them. Use the alphabetically first postal code.

We can join `scripts` and `practices` based on the fact that `'practice'` in `scripts` matches `'code'` in `practices`. However, we must first deal with the repeated values of `'code'` in `practices`. We want the alphabetically first postal codes.

In [ ]:

```
practice_postal = {}
for practice in practices:
    if practice['code'] in practice_postal:
        practice_postal[practice['code']] = ...
    else:
        practice_postal[practice['code']] = ...
```

Challenge: This is an aggregation of the practice data grouped by practice codes. Write an alternative implementation of the above cell using the `group_by_field` function you defined previously.

In [ ]:

```
assert practice_postal['K82019'] == 'HP21 8TR'
```

Challenge: This is an aggregation of the practice data grouped by practice codes. Write an alternative implementation of the above cell using the `group_by_field` function you defined previously.

In [ ]:

```
assert practice_postal['K82019'] == 'HP21 8TR'
```

Now we can join `practice_postal` to `scripts`.

In [ ]:

```
joined = scripts[:]
for script in joined:
    script['post_code'] = ...
```

Finally we'll group the prescription dictionaries in `joined` by `'post_code'` and sum up the items prescribed in each group, as we did in the previous question.

In [ ]:

```
items_by_post = ...
```

Question 4: items_by_region¶

Now we'll combine the techniques we've developed to answer a more complex question. Find the most commonly dispensed item in each postal code, representing the results as a list of tuples (`post_code`, `bnf_name`, amount dispensed as proportion of total). Sort your results ascending alphabetically by post code and take only results from the first 100 post codes.

NOTE: We'll continue to use the `joined` variable we created before, where we've chosen the alphabetically first postal code for each practice. Additionally, some postal codes will have multiple `'bnf_name'` with the same number of items prescribed for the maximum. In this case, we'll take the alphabetically first `'bnf_name'`.

Now we need to calculate the total items of each `'bnf_name'` prescribed in each `'post_code'`. Use the techniques we developed in the previous questions to calculate these totals. You should have 141196 `('post_code', 'bnf_name')` groups.

In [ ]:

```
total_items_by_bnf_post = ...
assert len(total_items_by_bnf_post) == 141196
```

Let's use `total_items` to find the maximum item total for each postal code. To do this, we will want to regroup `total_items_by_bnf_post` by `'post_code'` only, not by `('post_code', 'bnf_name')`. First let's turn `total_items` into a list of dictionaries (similar to `scripts` or `practices`) and then group it by `'post_code'`. You should have 118 groups in the resulting `total_items_by_post` after grouping `total_items` by `'post_code'`.

In [ ]:

```
total_items = ...
assert len(total_items_by_post) == 118
```

Now we will aggregate the groups in `total_by_item_post` to create `max_item_by_post`. Some `'bnf_name'` have the same item total within a given postal code. Therefore, if more than one `'bnf_name'` has the maximum item total in a given postal code, we'll take the alphabetically first `'bnf_name'`. We can do this by sorting each group according to the item total and `'bnf_name'`.

In [ ]:

```
max_item_by_post = ...
```

In order to express the item totals as a proportion of the total amount of items prescribed across all

`'bnf_name'` in a postal code, we'll need to use the total items prescribed that we previously calculated as `items_by_post`. Calculate the proportions for the most common `'bnf_names'` for each postal code. Format your answer as a list of tuples: `[(post_code, bnf_name, total)]`

In [ ]:

```
items_by_region = [('B11 4BW', 'Salbutamol_Inha 100mcg (200 D) CFF', 0.0341508247)] * 100
```

Apendix

Jupyter notebooks examples in this book are copyrighted: