Data Manipulation and Analysis with Python
Session 2

Robert Palmere

2021

Topics:

- Standard Python
- NumPy library
- Pandas library
- SymPy Applications

In this session we will cover some of the primary packages for numeric / symbolic calculations and data handling available in Python. In addition to introducing these modules, we will review some fundamentals of Python that we touched on in the first session.

This document serves as a guide for its associated Jupyter Notebook.

To start we will use a dataset regarding the attributes of benign and mallignant breast cancer cell nuclei from the 'sklearn' module and create a text file to contain these data.

```
from sklearn.datasets import load_breast_cancer
   def generate_data():
      X, y = load_breast_cancer(return_X_y=True)
      avg_radius = X[:, 0]
      avg_num_concaves = X[:, 7]
      lines = list(zip(avg_radius.astype(str),
          avg_num_concaves.astype(str)))
      lines = [' '.join(x) for x in lines]
      with open('Data.txt', 'w') as fp:
         fp.writelines('\n'.join(lines))

   generate_data()
```

We now have the data we will work with in "Data.txt" within our current working directory. We'll move on to retrieving this data using standard Python functions.

```
data = open('Data.txt', 'r')
```

The *open()* function of Python returns a wrapper class which containts methods to access the contents of a file using read mode ('r'). If we print the type of the object *data* we will see this explicitly.

```
print(type(data))
```

If one wanted to investigate what other methods are available from this returned class we can us the *dir()* method to list them:

```
for method in dir(data): print(f"'{method}'", end=' ')
```

Looking through these methods, *readlines()* seems like a reasonable choice to aquire our data for further analysis. We can inspect this method to see what arguments it access. Let's try it and then view the type it returns and examine if the developers included a *repr* to explain the details of this method.

```
lines = data.readlines()
print(type(lines))
print(repr(data.readlines))
```

Well, nothing is returned by *repr* to describe the function we called, but we see that it returned a list. Since it is a list we can use slicing to print the first few entries to examine the contents.

```
print(lines[0:5])
```

The entries are all string literals of the data found in "Data.txt". For analysis, we want this data to be in numeric form and we also want to split the lines into two separate columns.

```
for i in range(5): print(lines[i].split())
```

There are a few ways to cast these elements from a string to a float, in this case we will use the *map()* method. The map method takes in a function to cast each element of an iterable and the second argument is the iterable, which in this case is the list.

```
for i in range(5): print(map(float, lines[i].split()))
```

We will see that a map object is returned. A map object is just an iterator which we can convert to a list again using the *list()* method.

```
for i in range(5): print(list(map(float, lines[i].split())))
```

Let's separate the two columns of data representing the average radius and number of concavities of the nuclei into two separate lists.

```
avg_radius = [list(map(float, line.split()))[0] for line in lines]
avg_concavities = [list(map(float, line.split()))[1] for line in
    lines]
```

Now the data is ready for analysis and manipulation. We can place all of what we have done above into a single method in case we want to use it again.

```
def retrieve_data(filename):
   file = open(filename, 'r')
   lines = file.readlines()
   xs = [list(map(float, line.split()))[0] for line in lines]
      ys = [list(map(float, line.split()))[1] for line in lines]
   file.close()
      return xs, ys

x, y = retrieve_data('Data.txt')
```

Some simple manipulations of these data are presented in the Jupter notebook. If we desire to normalize the data and write the output, for instance, we can again use the *open()* method and *zip()* the contents of x, y into a single, comma separated, list again.

```
def output_normalized(x_norm, y_norm):
   lines = list(zip(list(map(str, x_norm)), list(map(str,
       y_norm))))
   lines = [' '.join(x) for x in lines]
```

```
    with open('Output.txt', 'w') as fp:
       fp.writelines('\n'.join(lines))
```

Many of the same manipulations we performed are part of the NumPy library. Not only is NumPy useful in the methods it provides, but also for increasing runtime speeds. Below, a speed comparison is performed by iterating over a list or NumPy array for 1000 iterations. However, if the list is not too large, the speeds are comparable.

```
import numpy as np
from timeit import default_timer as timer

def speedtest(key, length):
   d = {'array' : np.array([x for x in range(length)]),
       'list' : [x for x in range(length)]}
   time = []
   if key == list(d.keys())[0]:
      for i in range(10000):
         start = timer()

         mult = d['array'] * d['array']

         end = timer()
         dt = end - start
         time.append(dt)
   elif key == list(d.keys())[1]:
      for i in range(10000):
         start = timer()

         for n, i in enumerate(d['list']):
            mult = d['list'][n] * d['list'][n]

         end = timer()
         dt = end - start
         time.append(dt)
   return time

t = speedtest('list', 1000)
t2 = speedtest('array', 1000)

import matplotlib.pyplot as plt
plt.plot(t)
plt.plot(t2)
plt.ylabel('Time (s)')
plt.xlabel('Iteration')
plt.text(400, np.max(t)/2, s="Numpy is a lot faster.");
```

NumPy also provideees a few ways of reading in data from a file and can be rather convenient if we working with different file formats.

```
f = 'Data.txt'
data = np.loadtxt(f)
data = np.genfromtxt(f)
```

We can then use slicing again without list comprehension to separate the x and y values into two separate lists (or arrays in this case).

```
x = np.asarray(data[:, 0])
y = np.asarray(data[:, 1])
```

As a side note, the documentation strings we provide underneath functions within the Jupter notebook are accessible using the *doc* special method.

```
def example():
   '''Super helpful doc string.'''

example.__doc__

def doc(func):
   if callable(func):
      return func.__doc__
   else:
      raise ValueError('Argument must be a function.')

doc(example)
```

NumPy arrays are rather convenient when using operators as the operation is carried through the entire NumPy array without the need for a loop.

```
t1 = np.asarray([[1, 2], [3, 4]])
t2 = np.asarray([[5, 6], [7, 8]])

t2 > t1
```

The result of the above code is a boolean array with elements being true or false depending on whether or not the condition was satisfied pairwise between the two arrays. Additional convenience functions for array manipulations are presented in the Jupter notebook. Examples for use of the Pandas library are also shown. Pandas is similar to an excel sheet with automated indexing and

titles present for what would otherwise be a NumPy array.
The SymPy (Symbolic Python) gives the Python programmer access to methods for analytical mathematics similar to that provided by other software packages such as *Mathematica*. Symbols can be defined using the *symbols()* method and used to write analytical expressions.

```
from sympy import *
x, y, t = symbols('x y t')
print(x, y, t)
```

First we define x, y, and t to be symbols. The derivative can be taken with respect to x for a function like sin(x).

```
diff(sin(x)) # prints cos(x)
```

Two applications where SymPy might be useful for the study of chemical systems are also provided.