Software Carpentry Python Summary Document

Blair Wang
hello@blair.wang

September 19, 2020

This document is written to summarise (and, in some cases, extend) the "Programming with Python" Software Carpentry course. Each of the twelve parts of that course are summarised in this document in one A4 page.

The original materials for that Software Carpentry course can be found online at `https://swcarpentry.github.io/python-novice-inflammation/`. The source code for this document can be found online at `https://github.com/blairw/swcpythonsummary`.

Contents

1 Python Fundamentals

1.1 Variables

To start with, we can think of Python as a calculator so you can do a calculation like "3 + 5 × 4":

```
>>> 3 + 5 * 4
23
```

The above is what it looks like in an interactive Python shell (`python3` at a bash/zsh shell). However, if you are running a Python *script* (e.g. `python3 myscript.py`), you will not see anything. In that latter case, you will need to type into your Python script:

```
print(3 + 5 * 4)
```

Like any good 'scientific' calculator, you can also store the results of calculations into variables like how you can do `Ans` → `A`. However, unlike your typical high school scientific calculator, Python gives you much more flexibility in naming your variables. Therefore, you should give your variables meaningful names. `A` would actually be a very bad name for a variable in most cases. Your variables should be named according to the business context, e.g. if we were writing some Python script about the Consumer Price Index (CPI):

```
cpi_june_2019 = 0.6
print(cpi_june_2019)
```

1.2 Data Types

You will notice in the above we first worked with **integers** ($\mathbb{Z}$, e.g. 1, 2, 3) and then there was an example with a **rational number** ($\mathbb{Q}$, e.g. 0.6, $\frac{1}{3}$). Rational numbers are stored in Python as so-called "floating-point" numbers or "floats". Floats can sometimes behave strangely:

```
>>> 1.1 + 1.3
2.4000000000000004
```

To cope with this, we can use **libraries**. A library is some additional component of Python that gives us access to additional programming functionality, e.g.:

```
>>> from fractions import Fraction
>>> from decimal import Decimal
>>>
>>> Decimal(1.1) + Decimal(1.3)
Decimal('2.400000000000000133226762955')
>>> # this ^ is still the wrong answer
>>> # we should enclose the numbers in quotes
>>>
>>> Decimal('1.1') + Decimal('1.3')
Decimal('2.4')
>>> # correct answer! :)
>>>
>>> Fraction(1,3) + Fraction(1,3)
Fraction(2, 3)
>>> print(Fraction(1,3) + Fraction(1,3))
2/3
>>> print(float(Fraction(1,3) + Fraction(1,3)))
0.6666666666666666
```

You may notice in the above that I have used the "hash" symbol to make **comments**. This is very useful for explaining your code.

1.3 Additional Tips

- If you want to, you can assign multiple variables at the same time, e.g. `mass, age = 25.5, 20`. Only do this if it makes your code more readable.
- In additional to the numerical data types discussed above, Python also has the data type for text strings (e.g. the classic: `print('Hello world')`). You can use either single quotes or double quotes. This can be useful in some situations, e.g. `print('In French: "Bonjour tout le monde"')`.
- Variable names can include letters, digits and underscores. However, they cannot *begin* with a digit. For example, you cannot create a variable called `12th_person`.

2 Analyzing Patient Data

2.1 Loading the "Inflammation" CSV Files

The Software Carpentry Python course involves a very simple case study involve a set of CSV (comma-separated values) data files. CSV files are simply text files where each line of text is a new role and each line is formatted like `0.5,52.2,12.1` (hence *comma-separated* values). In our "Inflammation" case study, which is from a medical context, each row is an arthritis patient and each column is a day of data about the severity of inflammation that these patients are experiencing.

Because this is a CSV file that is all numerical, we can use the numbers library for Python (`numpy`) to load the CSV file into a **2D array**. A array is simply a list of things, so a 2D array is a list of lists. This can represent a table structure like a CSV file: a list of row data, and each row datum is a list of values across the columns.

```
import numpy as np
data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')
print(data.shape) # returns (60, 40) = 60 rows, 40 columns
```

Please note that this will not work unless you have numpy installed. I recommend doing this using a virtual environment (venv):

```
python3 -m venv .venv           # create the venv
source .venv/bin/activate       # activate the venv
python3 -m pip install numpy    # install numpy inside the venv
python3                         # do stuff in Python
deactivate                      # deactivate the venv when done
```

2.2 Getting Data Points and Ranges

- Arrays in Python are indexed from 0. This means that, for example, that what we would usually call the "first" column is, in Python, column "0"; the "second" column is column "1", etc.
- You can get a data point ("cell" in spreadsheet terminology) by **specifying coordinates as "row, column"**. For example, what we might think of in Microsoft Excel as cell "C5" (= "fifth row, third column" = row "4", column "2" in Python) can be obtained like so:

```
data_c5 = data[4, 2]
```

- You can get a range by specifying the starting point and that which is *after* the ending point. This could be quite intuitive in the sense that `data[0:4, 0:10]` means "first 4 rows, first 10 columns", but since you can specify ranges that don't start from 0, it could also take some more mental arithmetic to be sure what you're doing (e.g. `data[5:8, 3:15]` = rows 5 to 7 (0-indexed), columns 3 to 14 (0-indexed)).
- You can also have unbounded ranges, e.g. `data[ :8, 3: ]` = all rows up to and including row 7, all rows including and after column 3.

2.3 Descriptive Statistics

Given a numpy array, you can print the usual descriptive statistics:

```
print('minimum inflammation:', np.min(data))
print('maximum inflammation:', np.max(data))
print('mean:', np.mean(data))
print('standard deviation:', np.std(data))
```

Each of those operations (min, max, mean, std) returns a signle value. However, you could also generate an array of values. For example, maybe you want a list of the mean inflammation scores for each patient (i.e., mean across rows); or you might want a list of mean inflammation scores for each day (i.e., mean across columns).

To do this, we specify a value for `axis`:

```
avg_across_days = numpy.mean(data, axis=0)
avg_for_each_patient = numpy.mean(data, axis=1)
```

You basically just have to remember that **"axis 0" means columns and "axis 1" means rows**, even though coordinates are specified as "row, column".

3 Visualizing Tabular Data

The Software Carpentry Python course suggests using the `matplotlib` library for charts and graphs. This is a decent library for simple charts and graphs, though for anything more sophisticated you might want to look into `seaborn`.

Important Reminder: Just like with `numpy`, you will have to install `matplotlib` before proceeding:

```
deactivate && source .venv/bin/activate
python3 -m pip install matplotlib
```

3.1 Heatmap with `imshow`

Assuming we still have `data` set up as the CSV numpy array from the previous section, we can generate a heatmap simply with:

```
1 import matplotlib.pyplot
2 image = matplotlib.pyplot.imshow(data)
3 matplotlib.pyplot.savefig('imshow_output.png')
4 matplotlib.pyplot.show()
```

Note that the line 4 only works if you are running Python in a graphical environment: it shows a pop-up with the graphic. **This is not the case if you are running Python in a Linux shell inside WSL 1.** In that situation, you can remove line 4 and just use the `imshow_output.png` file generated in line 3.

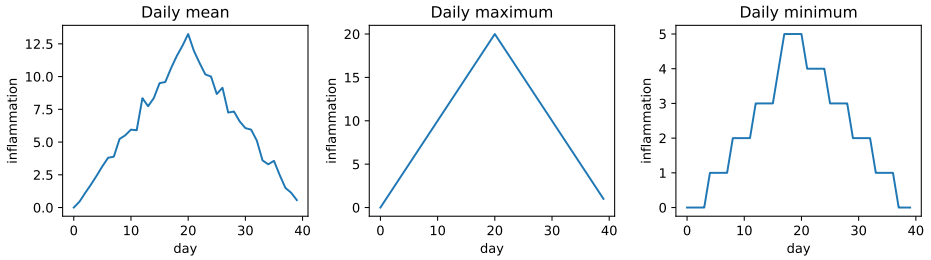
3.2 Graphing with `plot`

The `matplotlib` library also draws simple line graphs. For example, recalling that each column in this dataset is a day's worth of data, **"axis 0" means columns**, we could graph the trend of the daily average inflammation:

```
1 ave_inflammation = numpy.mean(data, axis=0)
2 ave_plot = matplotlib.pyplot.plot(ave_inflammation)
3 matplotlib.pyplot.savefig('plot_output.png')
```

3.3 Drawing a `figure`

You might recall some rules about proper charts and graphs: you should always include a title; you should always label the axes. You may also wish to combine multiple charts and graphs into a single picture. All the above can be achieved using `figure`.




```
1  fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
2
3  axes1 = fig.add_subplot(1, 3, 1); axes1.set_title('Daily mean')
4  axes1.set_xlabel('day'); axes1.set_ylabel('inflammation')
5  axes1.plot(numpy.mean(data, axis=0))
6
7  axes2 = fig.add_subplot(1, 3, 2); axes2.set_title('Daily maximum')
8  axes2.set_xlabel('day'); axes2.set_ylabel('inflammation')
9  axes2.plot(numpy.max(data, axis=0))
10
11 axes3 = fig.add_subplot(1, 3, 3); axes3.set_title('Daily minimum')
12 axes3.set_xlabel('day'); axes3.set_ylabel('inflammation')
13 axes3.plot(numpy.min(data, axis=0))
14
15 fig.tight_layout()
16 matplotlib.pyplot.savefig('inflammation_figure.pdf')
```

Notice that we saved the image as PDF file on line 16. This is exceptionally useful since the resulting PDF file is a vector graphic, not a bitmap. I have actually included this image above - try selecting "daily maximum" with your mouse, or try zooming in and seeing how there is no "pixellation".

4 Repeating Actions with Loops

Consider the example from the previous section in which we created `axes1`, `axes2` and `axes3` and placed them into a `matplotlib` figure. If you look at the code for `axes1`, `axes2` and `axes3` (lines 3 to 5, 7 to 9, and 11 to 13), you'll notice they are all quite similar. The only things that differ for each of these is:

- The third parameter for the `fig.add_subplot()` function;
- the title of the graph; and,
- the data source for the graph.

Everything else about the graphs was the same. This is a great example of when we could use a **for-loop**. The code sample below **refactors** *(= rewriting code to make it more elegant, while retaining the same outcome of the code)* the previous code sample to use a for-loop.

```
1  fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))
2
3  figure_data = [
4      {'title': 'Daily mean',      'numbers': numpy.mean(data, axis=0)}
5      , {'title': 'Daily maximum', 'numbers': numpy.max(data, axis=0)}
6      , {'title': 'Daily minimum', 'numbers': numpy.min(data, axis=0)}
7  ]
8
9  for seqno in range(0, 3):
10     this_axes = fig.add_subplot(1, 3, seqno + 1)
11     this_axes.set_xlabel('day')
12     this_axes.set_ylabel('inflammation')
13     this_axes.set_title(figure_data[seqno]['title'])
14     this_axes.plot(figure_data[seqno]['numbers'])
15
16 fig.tight_layout()
17 matplotlib.pyplot.savefig('inflammation_figure.pdf')
```

A few things to note here:

- Line 1 is the same as it was before.
- On lines 3 to 7, we now have a data structure called `figure_data`. This is an array with two elements, each of which is in the format `{ 'k1': v1, 'k2': v2 }`. This is known as a **dictionary**. It allows us to store **values** (e.g. *k1, k2*) for specified **keys** (e.g. *v1, v2*). So, for example, the value stored at `figure_data[0]['title']` is the string *"Daily mean"*.
- On lines 9 to 14, we now have a control structure called a **for-loop**. This means that everything "inside" the loop (lines 10-14) is executed in each iteration of the loop. **In Python, we indicate what is "inside" the loop using indentation, which is why lines 10-14 are indented.** This is very important if you are coming from a language like Java where indentation is not as consequential.
- On line 9, we define the for-loop using `range(0, 3)`. This is a function that generates an array from 0 (inclusive) to 3 (exclusive), i.e., {*0, 1, 2*}. The loop iterates across each of these elements, with each iteration storing the element as `seqno`. This conveniently means that while we traverse `range(0, 3)`, we also traverse the elements of `figure_data`, which becomes very useful on lines 13 and 14. It also means, technically, we could have used `range(0, len(figure_data))` instead of `range(0, 3)`.
- However, because `fig.add_subplot()` expects {*1, 2, 3*} for its third parameter (i.e. 1-indexed instead of 0-indexed), we have the expression `seqno + 1` on line 10.
- On line 9, we could have alternatively used the expression `for seqno, this_value in enumerate(figure_data)`. This would allow us to replace `figure_data[seqno]` with simply `this_value` on lines 13 and 14. This is similar to a **for-each loop** (which you may be familiar with if you have a Java background).

You should also know that you can treat a string as an array of one-character strings, and use a for-loop on it likewise:

```
1  word = 'oxygen'
2  for char in word:
3      print(char)
```