07_Creating_Functions

August 12, 2021

1 Creating Functions

1.1 Questions

- How can I define new functions?
- What's the difference between defining and calling a function?
- What happens when I call a function?

1.2 Objectives

1. Define a function that takes parameters.
2. Return a value from a function.
3. Test and debug a function.
4. Set default values for function parameters.
5. Explain why we should divide programs into small, single-purpose functions.

1.3 Credit:

things here are a mix of the really excellent Software Carpentry tutorial on Python: http://swcarpentry.github.io/python-novice-inflammation/

I've made some slight adaptations here and there, but the credit goes to those organizations. I hope I am using this correctly under the licences:

https://earth-env-data-science.github.io/LICENSE.html https://swcarpentry.github.io/python-novice-inflammation/LICENSE.html

---

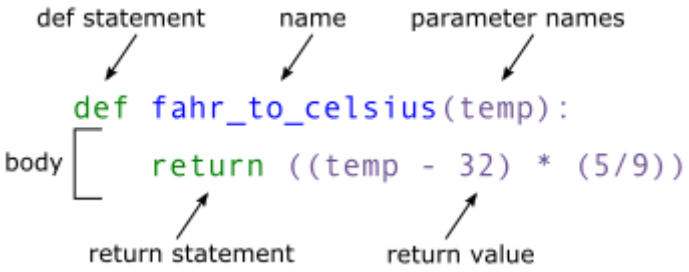
At this point, we've written code to: * draw some interesting features in our inflammation data, * loop over all our data files to quickly draw these plots for each of them, * and have Python make decisions based on what it sees in our data.

But, our code is getting pretty long and complicated; what if we had thousands of datasets, and didn't want to generate a figure for every single one? Commenting out the figure-drawing code is a nuisance. Also, what if we want to use that code again, on a different dataset or at a different point

in our program? Cutting and pasting it is going to make our code get very long and very repetitive, very quickly. We'd like a way to package our code so that it is easier to reuse, and Python provides for this by letting us define things called 'functions' - a shorthand way of re-executing longer pieces of code.

Let's start by defining a function `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

[1]:
```
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
```



The function definition opens with the keyword `def` followed by the name of the function (`fahr_to_celsius`) and a parenthesized list of parameter names (`temp`). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a `return` keyword followed by the return value.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Let's try running our function. Calling our own function is no different from calling any other function:

[2]:
```
fahr_to_celsius(32)
```

[2]:
```
0.0
```

[3]:
```
print('freezing point of water:', fahr_to_celsius(32))
print('boiling point of water:', fahr_to_celsius(212))
```

```
freezing point of water: 0.0
boiling point of water: 100.0
```

We've successfully called the function that we defined, and we have access to the value that we returned.

1.4 Composing Functions

Now that we've seen how to turn Fahrenheit into Celsius, we can also write the function to turn Celsius into Kelvin:

[4]:
```
def celsius_to_kelvin(temp_c):
    return temp_c + 273.15

print('freezing point of water in Kelvin:', celsius_to_kelvin(0.))
```

```
freezing point of water in Kelvin: 273.15
```

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

[5]:
```
def fahr_to_kelvin(temp_f):
    temp_c = fahr_to_celsius(temp_f)
    temp_k = celsius_to_kelvin(temp_c)
    return temp_k

print('boiling point of water in Kelvin:', fahr_to_kelvin(212.0))
```

```
boiling point of water in Kelvin: 373.15
```

1.4.1 key note:

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-larger chunks to get the effect we want. Real-life functions will usually be larger than the ones shown here — typically half a dozen to a few dozen lines — but they shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on.

2 Tidying Up

Now that we know how to wrap bits of code up in functions, we can make our inflammation analysis easier to read and easier to reuse. First, let's make an `analyze` function that generates our plots:

[6]:
```
def analyze(filename):
    data = np.loadtxt(filename, delimiter=',')

    fig = plt.figure(figsize=(5, 5))
    ax = fig.add_subplot(1, 1, 1)

    ax.plot(np.mean(data, axis=0), label="mean")
    ax.plot(np.max(data, axis=0), label="max")
    ax.plot(np.min(data, axis=0), label="min")

    ax.set_ylabel('Inflammation')
    ax.set_xlabel('Days')

    ax.legend()
```

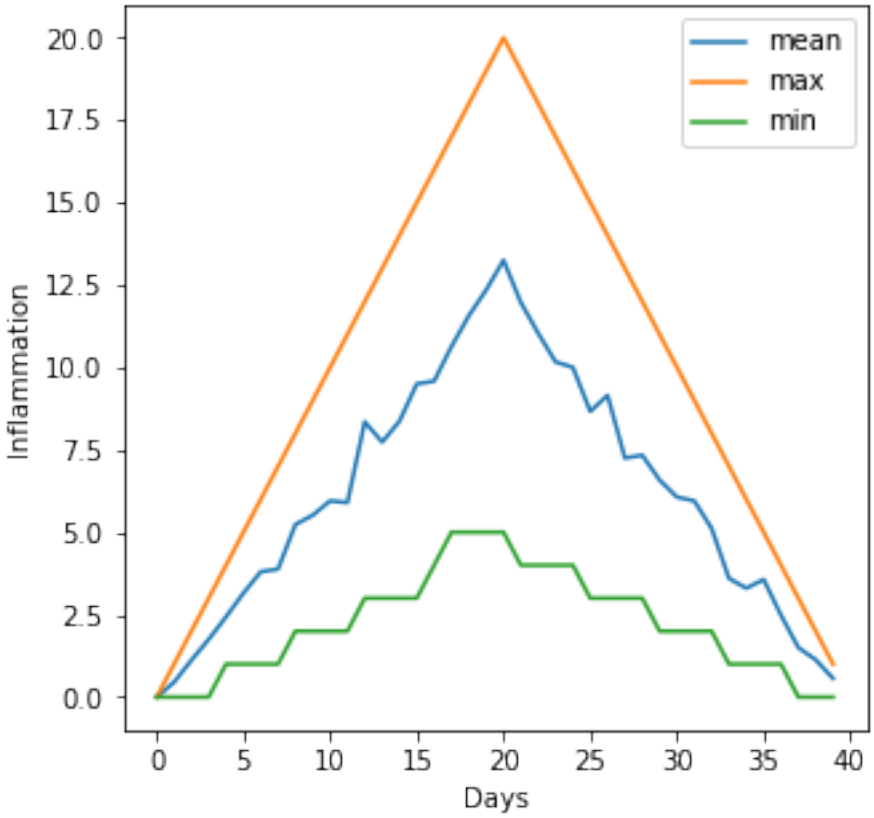
Notice that rather than jumbling this code together in one giant `for` loop, we can now reproduce the previous analysis with a much simpler `for` loop. First we need to import those libraries we've

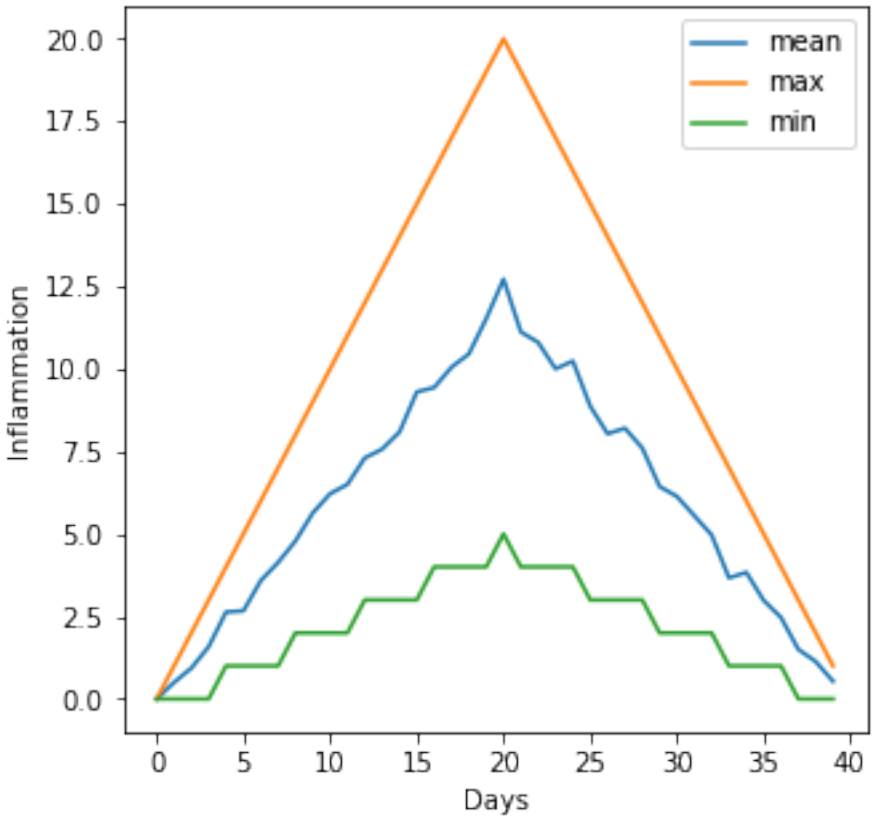
been using:

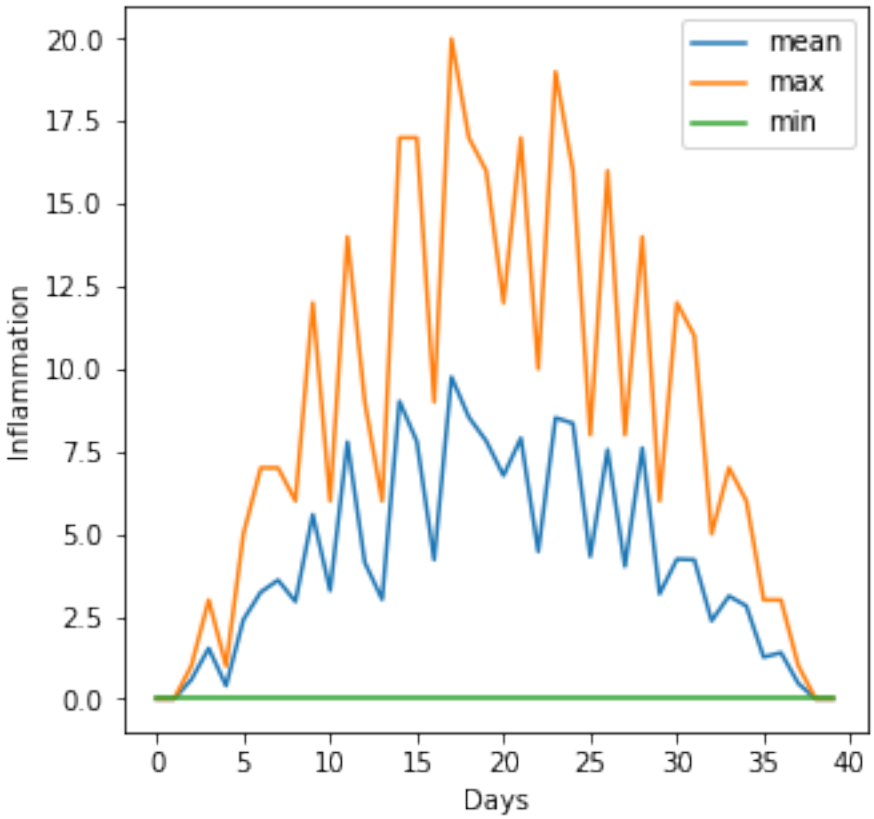
[7]:
```
import numpy as np
import matplotlib.pyplot as plt
import glob

%matplotlib inline
```

[8]:
```
# read in the filenames using glob:

filenames = sorted(glob.glob('../data/inflammation*.csv'))

#create a 'for' loop that prints the filename, runs analyze():
for f in filenames[:3]:
    print(f)
    analyze(f)
```

```
../data/inflammation-01.csv
../data/inflammation-02.csv
../data/inflammation-03.csv
```




mean
max
min
Inflammation
Days
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20
25
30
35
40



By giving our functions human-readable names, we can more easily read and understand what is happening in the for loop. Even better, if at some later date we want to use either of those pieces of code again, we can do so in a single line.

3 Documenting our function

The usual way to put documentation in software is to add comments within our code like this:

[9]:
```
# analyze(filename):
# load a file and plot the mean, max and min of the patient inflammation data␣
 ↪over time:
def analyze(filename):
    data = np.loadtxt(filename, delimiter=',')

    fig = plt.figure(figsize=(5, 5))
    ax = fig.add_subplot(1, 1, 1)

    ax.plot(np.mean(data, axis=0), label="mean")
    ax.plot(np.max(data, axis=0), label="max")
    ax.plot(np.min(data, axis=0), label="min")
```

```
    ax.set_ylabel('Inflammation')
    ax.set_xlabel('Days')

    ax.legend()
```

There's a better way, though.We can write a special string with three `'''` to create documentation that goes with the function:

[10]:
```
def analyze(filename):
    '''load a file and plot the mean, max, and min of the data through time '''

    data = np.loadtxt(filename, delimiter=',')

    fig = plt.figure(figsize=(5, 5))
    ax = fig.add_subplot(1, 1, 1)

    ax.plot(np.mean(data, axis=0), label="mean")
    ax.plot(np.max(data, axis=0), label="max")
    ax.plot(np.min(data, axis=0), label="min")

    ax.set_ylabel('Inflammation')
    ax.set_xlabel('Days')

    ax.legend()
```

How would we look up the documentation? With `help`

[11]:
```
help(analyze)
```

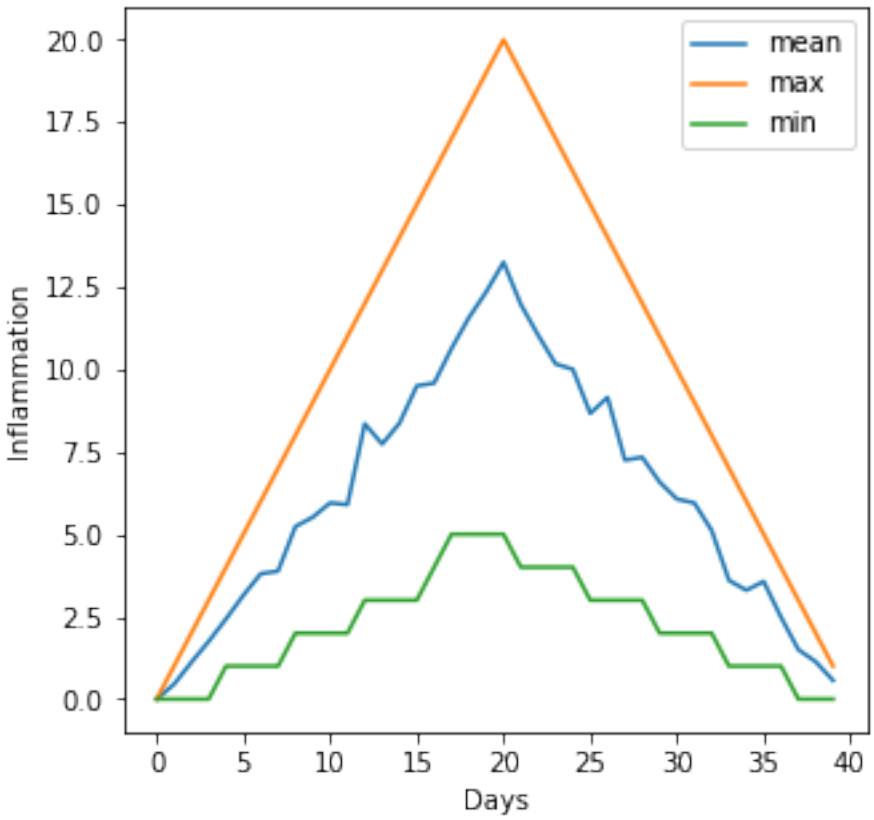
```
Help on function analyze in module __main__:

analyze(filename)
    load a file and plot the mean, max, and min of the data through time
```

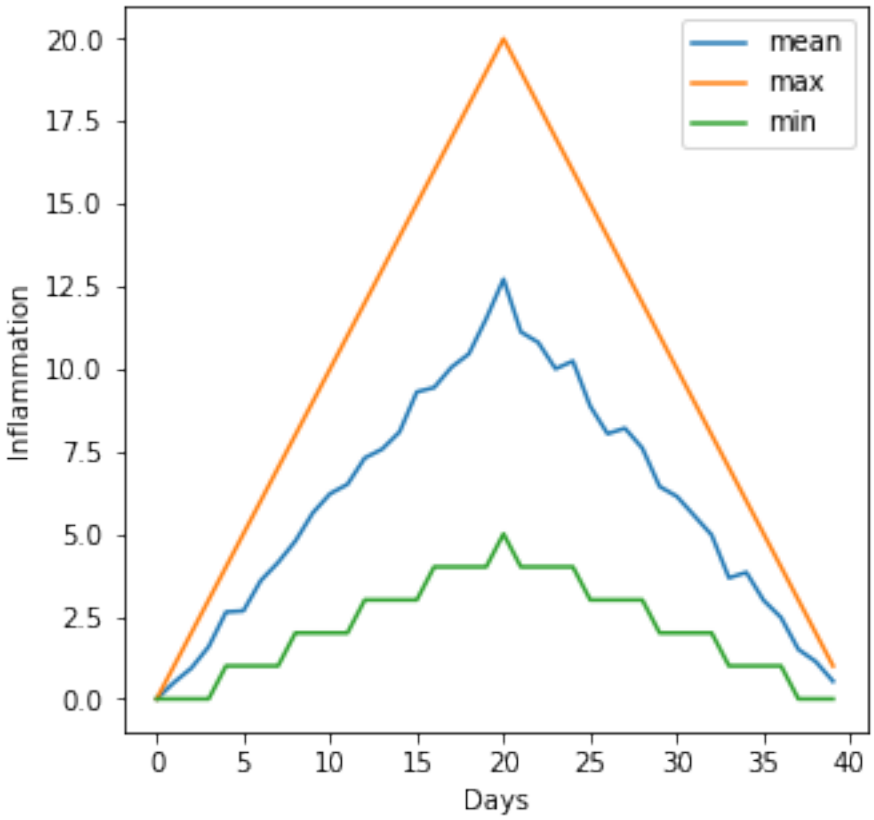
4 How to use this function elsewhere?

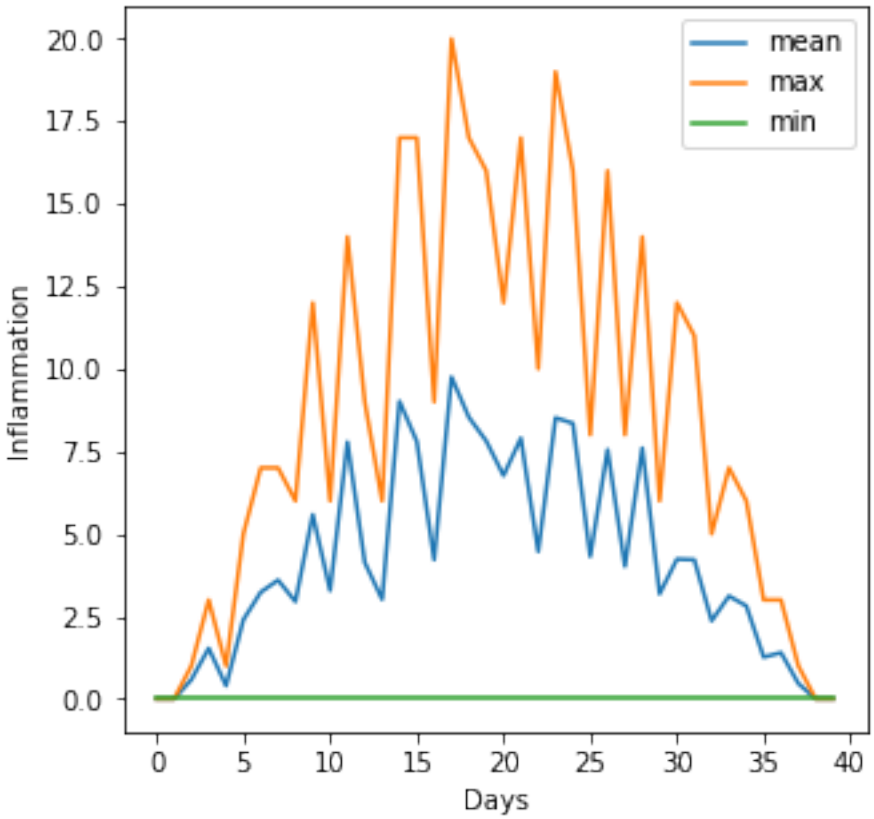
4.0.1 make sure to do: settings, text editor indentation, 4 space

[12]:
```
from helpers import analyze2
import glob

filenames = sorted(glob.glob('../data/inflammation*.csv'))

for f in filenames[:3]:
    print(f)
    analyze2(f)
```

```
../data/inflammation-01.csv
../data/inflammation-02.csv
../data/inflammation-03.csv
```




mean
max
min
Inflammation
Days
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20
25
30
35
40



This is better because we can now ask Python's built-in help system to show us the documentation for the function. This means that no matter where we 'are' we can ask for the information about the fucntion we wrote (and obviously can't remember how to use):

4.1 Readable functions

Consider these two functions:

4.1.1 Function 1

```
def s(p):
    a = 0
    for v in p:
        a += v
    m = a / len(p)
    d = 0
    for v in p:
        d += (v - m) * (v - m)
    return numpy.sqrt(d / (len(p) - 1))
```

4.1.2 Function 2

```
def std_dev(sample):
    sample_sum = 0
    for value in sample:
        sample_sum += value

    sample_mean = sample_sum / len(sample)

    sum_squared_devs = 0
    for value in sample:
        sum_squared_devs += (value - sample_mean) * (value - sample_mean)

    return numpy.sqrt(sum_squared_devs / (len(sample) - 1))
```

The functions s and std_dev are computationally equivalent (they both calculate the sample standard deviation), but to a human reader, they look very different. You probably found std_dev much easier to read and understand than s.

As this example illustrates, both documentation and a programmer's coding style combine to determine how easy it is for others to read and understand the programmer's code. Choosing meaningful variable names and using blank spaces to break the code into logical "chunks" are helpful techniques for producing readable code. This is useful not only for sharing code with others, but also for the original programmer. If you need to revisit code that you wrote months ago and haven't thought about since then, you will appreciate the value of readable code!

---

5 Key Points

Define a function using `def function_name(parameter)`.

The body of a function must be indented.

Call a function using `function_name(value)`.

Use `help(thing)` to view help for something.

Put `docstrings` in functions to provide help for that function.