‹ (/python-novice-inflammation/07-cond/index.html) Programming with Python (/python-novice-inflammation/) › (/pyth novice inflam errors

Creating Functions

Overview

Teaching: 30 min
Exercises: 0 min
Questions

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

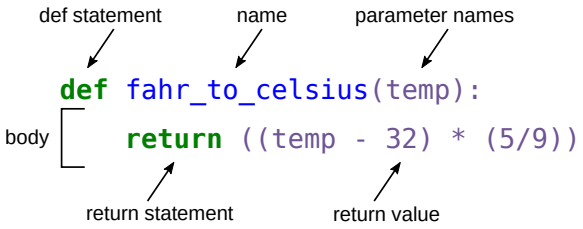
Objectives

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

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_celsius` that converts temperatures from Fahrenheit to Celsius:

Python

```
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 (../reference.html#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 (../reference.html#return-statement) to send a result back to whoever asked for it.

Let's try running our function.

Python

```
fahr_to_celsius(32)
```

This command should call our function, using "32" as the input and return the function value.

In fact, calling our own function is no different from calling any other function:

Python

```
print('freezing point of water:', fahr_to_celsius(32), 'C')
print('boiling point of water:', fahr_to_celsius(212), 'C')
```

Output

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

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

Composing Functions

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

Python

```
def celsius_to_kelvin(temp_c):
    return temp_c + 273.15

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

Output

```
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 (../reference.html#compose) the two functions we have already created:

Python

```
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))
```

Output

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

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.

Variable Scope

In composing our temperature conversion functions, we created variables inside of those functions, `temp`, `temp_c`, `temp_f`, and `temp_k`. We refer to these variables as local variables (../reference.html#local-variable) because they no longer exist once the function is done executing. If we try to access their values outside of the function, we will encounter an error:

Python

```
print('Again, temperature in Kelvin was:', temp_k)
```

Error

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-eed2471d229b> in <module>
----> 1 print('Again, temperature in Kelvin was:', temp_k)

NameError: name 'temp_k' is not defined
```

If you want to reuse the temperature in Kelvin after you have calculated it with `fahr_to_kelvin`, you can store the result of the function call in a variable:

Python

```
temp_kelvin = fahr_to_kelvin(212.0)
print('temperature in Kelvin was:', temp_kelvin)
```

Output

```
temperature in Kelvin was: 373.15
```

The variable `temp_kelvin`, being defined outside any function, is said to be global (../reference.html#global-variable).

Inside a function, one can read the value of such global variables:

Python

```
def print_temperatures():
  print('temperature in Fahrenheit was:', temp_fahr)
  print('temperature in Kelvin was:', temp_kelvin)

temp_fahr = 212.0
temp_kelvin = fahr_to_kelvin(temp_fahr)

print_temperatures()
```

Output

```
temperature in Fahrenheit was: 212.0
temperature in Kelvin was: 373.15
```

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 a `visualize` function that generates our plots:

Python

```
def visualize(filename):

    data = numpy.loadtxt(fname=filename, delimiter=',')

    fig = matplotlib.pyplot.figure(figsize=(10.0, 3.0))

    axes1 = fig.add_subplot(1, 3, 1)
    axes2 = fig.add_subplot(1, 3, 2)
    axes3 = fig.add_subplot(1, 3, 3)

    axes1.set_ylabel('average')
    axes1.plot(numpy.mean(data, axis=0))

    axes2.set_ylabel('max')
    axes2.plot(numpy.max(data, axis=0))

    axes3.set_ylabel('min')
    axes3.plot(numpy.min(data, axis=0))

    fig.tight_layout()
    matplotlib.pyplot.show()
```

and another function called `detect_problems` that checks for those systematics we noticed:

Python

```
def detect_problems(filename):

    data = numpy.loadtxt(fname=filename, delimiter=',')

    if numpy.max(data, axis=0)[0] == 0 and numpy.max(data, axis=0)[20] == 20:
        print('Suspicious looking maxima!')
    elif numpy.sum(numpy.min(data, axis=0)) == 0:
        print('Minima add up to zero!')
    else:
        print('Seems OK!')
```

Wait! Didn't we forget to specify what both of these functions should return? Well, we didn't. In Python, functions are not required to include a `return` statement and can be used for the sole purpose of grouping together pieces of code that conceptually do one thing. In such cases, function names usually describe what they do, *e.g.* `visualize`, `detect_problems`.

Notice that rather than jumbling this code together in one giant `for` loop, we can now read and reuse both ideas separately. We can reproduce the previous analysis with a much simpler `for` loop:

Python

```
filenames = sorted(glob.glob('inflammation*.csv'))

for filename in filenames[:3]:
    print(filename)
    visualize(filename)
    detect_problems(filename)
```

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.

Testing and Documenting

Once we start putting things in functions so that we can re-use them, we need to start testing that those functions are working correctly. To see how to do this, let's write a function to offset a dataset so that it's mean value shifts to a user-defined value:

Python

```
def offset_mean(data, target_mean_value):
    return (data - numpy.mean(data)) + target_mean_value
```

We could test this on our actual data, but since we don't know what the values ought to be, it will be hard to tell if the result was correct. Instead, let's use NumPy to create a matrix of 0's and then offset its values to have a mean value of 3:

Python

```
z = numpy.zeros((2,2))
print(offset_mean(z, 3))
```

Output

```
[[ 3.  3.]
 [ 3.  3.]]
```

That looks right, so let's try `offset_mean` on our real data:

Python

```
data = numpy.loadtxt(fname='inflammation-01.csv', delimiter=',')
print(offset_mean(data, 0))
```

Output

```
[[-6.14875 -6.14875 -5.14875 ... -3.14875 -6.14875 -6.14875]
 [-6.14875 -5.14875 -4.14875 ... -5.14875 -6.14875 -5.14875]
 [-6.14875 -5.14875 -5.14875 ... -4.14875 -5.14875 -5.14875]
 ...
 [-6.14875 -5.14875 -5.14875 ... -5.14875 -5.14875 -5.14875]
 [-6.14875 -6.14875 -6.14875 ... -6.14875 -4.14875 -6.14875]
 [-6.14875 -6.14875 -5.14875 ... -5.14875 -5.14875 -6.14875]]
```

It's hard to tell from the default output whether the result is correct, but there are a few tests that we can run to reassure us:

Python

```
print('original min, mean, and max are:', numpy.min(data), numpy.mean(data), numpy.max(data))
offset_data = offset_mean(data, 0)
print('min, mean, and max of offset data are:',
      numpy.min(offset_data),
      numpy.mean(offset_data),
      numpy.max(offset_data))
```

Output

```
original min, mean, and max are: 0.0 6.14875 20.0
min, mean, and and max of offset data are: -6.14875 2.84217094304e-16 13.85125
```

That seems almost right: the original mean was about 6.1, so the lower bound from zero is now about -6.1. The mean of the offset data isn't quite zero — we'll explore why not in the challenges — but it's pretty close. We can even go further and check that the standard deviation hasn't changed:

Python

```
print('std dev before and after:', numpy.std(data), numpy.std(offset_data))
```

Output

```
std dev before and after: 4.61383319712 4.61383319712
```

Those values look the same, but we probably wouldn't notice if they were different in the sixth decimal place. Let's do this instead:

Python

```
print('difference in standard deviations before and after:',
      numpy.std(data) - numpy.std(offset_data))
```

Output

```
difference in standard deviations before and after: -3.5527136788e-15
```

Again, the difference is very small. It's still possible that our function is wrong, but it seems unlikely enough that we should probably get back to doing our analysis. We have one more task first, though: we should write some documentation (../reference.html#documentation) for our function to remind ourselves later what it's for and how to use it.

The usual way to put documentation in software is to add comments (../reference.html#comment) like this:

Python

```
# offset_mean(data, target_mean_value):
# return a new array containing the original data with its mean offset to match the desired value.
def offset_mean(data, target_mean_value):
    return (data - numpy.mean(data)) + target_mean_value
```

There's a better way, though. If the first thing in a function is a string that isn't assigned to a variable, that string is attached to the function as its documentation:

Python

```
def offset_mean(data, target_mean_value):
    """Return a new array containing the original data
       with its mean offset to match the desired value."""
    return (data - numpy.mean(data)) + target_mean_value
```

This is better because we can now ask Python's built-in help system to show us the documentation for the function:

Python

```
help(offset_mean)
```

Output

```
Help on function offset_mean in module __main__:

offset_mean(data, target_mean_value)
    Return a new array containing the original data with its mean offset to match the desired value.
```

A string like this is called a docstring (../reference.html#docstring). We don't need to use triple quotes when we write one, but if we do, we can break the string across multiple lines:

Python

```
def offset_mean(data, target_mean_value):
    """Return a new array containing the original data
       with its mean offset to match the desired value.

    Examples
    --------
    >>> offset_mean([1, 2, 3], 0)
    array([-1.,  0.,  1.])
    """
    return (data - numpy.mean(data)) + target_mean_value

help(offset_mean)
```

Output

```
Help on function offset_mean in module __main__:

offset_mean(data, target_mean_value)
    Return a new array containing the original data
       with its mean offset to match the desired value.

    Examples
    --------
    >>> offset_mean([1, 2, 3], 0)
    array([-1.,  0.,  1.])
```

Defining Defaults

We have passed parameters to functions in two ways: directly, as in `type(data)`, and by name, as in `numpy.loadtxt(fname='something.csv', delimiter=',')`. In fact, we can pass the filename to `loadtxt` without the `fname=`:

Python

```
numpy.loadtxt('inflammation-01.csv', delimiter=',')
```

Output

```
array([[ 0.,  0.,  1., ...,  3.,  0.,  0.],
       [ 0.,  1.,  2., ...,  1.,  0.,  1.],
       [ 0.,  1.,  1., ...,  2.,  1.,  1.],
       ...,
       [ 0.,  1.,  1., ...,  1.,  1.,  1.],
       [ 0.,  0.,  0., ...,  0.,  2.,  0.],
       [ 0.,  0.,  1., ...,  1.,  1.,  0.]])
```

but we still need to say `delimiter=`:

Python

```
numpy.loadtxt('inflammation-01.csv', ',')
```

Error

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/username/anaconda3/lib/python3.6/site-packages/numpy/lib/npyio.py", line 1041, in loa
dtxt
    dtype = np.dtype(dtype)
  File "/Users/username/anaconda3/lib/python3.6/site-packages/numpy/core/_internal.py", line 199, in
_commastring
    newitem = (dtype, eval(repeats))
  File "<string>", line 1
    ,
    ^
SyntaxError: unexpected EOF while parsing
```

To understand what's going on, and make our own functions easier to use, let's re-define our `offset_mean` function like this:

Python

```
def offset_mean(data, target_mean_value=0.0):
    """Return a new array containing the original data
       with its mean offset to match the desired value, (0 by default).

    Examples
    --------
    >>> offset_mean([1, 2, 3])
    array([-1.,  0.,  1.])
    """
    return (data - numpy.mean(data)) + target_mean_value
```

The key change is that the second parameter is now written `target_mean_value=0.0` instead of just `target_mean_value` . If we call the function with two arguments, it works as it did before:

Python

```
test_data = numpy.zeros((2, 2))
print(offset_mean(test_data, 3))
```

Output

```
[[ 3.  3.]
 [ 3.  3.]]
```

But we can also now call it with just one parameter, in which case `target_mean_value` is automatically assigned the default value (../reference.html#default-value) of 0.0:

Python

```
more_data = 5 + numpy.zeros((2, 2))
print('data before mean offset:')
print(more_data)
print('offset data:')
print(offset_mean(more_data))
```

Output

```
data before mean offset:
[[ 5.  5.]
 [ 5.  5.]]
offset data:
[[ 0.  0.]
 [ 0.  0.]]
```

This is handy: if we usually want a function to work one way, but occasionally need it to do something else, we can allow people to pass a parameter when they need to but provide a default to make the normal case easier. The example below shows how Python matches values to parameters:

Python

```
def display(a=1, b=2, c=3):
    print('a:', a, 'b:', b, 'c:', c)

print('no parameters:')
display()
print('one parameter:')
display(55)
print('two parameters:')
display(55, 66)
```

Output

```
no parameters:
a: 1 b: 2 c: 3
one parameter:
a: 55 b: 2 c: 3
two parameters:
a: 55 b: 66 c: 3
```

As this example shows, parameters are matched up from left to right, and any that haven't been given a value explicitly get their default value. We can override this behavior by naming the value as we pass it in:

Python

```
print('only setting the value of c')
display(c=77)
```

Output

```
only setting the value of c
a: 1 b: 2 c: 77
```

With that in hand, let's look at the help for `numpy.loadtxt`:

Python

```
help(numpy.loadtxt)
```

Output

```
Help on function loadtxt in module numpy.lib.npyio:

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, use
cols=None, unpack=False, ndmin=0, encoding='bytes')
    Load data from a text file.

    Each row in the text file must have the same number of values.

    Parameters
    ----------
...
```

There's a lot of information here, but the most important part is the first couple of lines:

Output

```
loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, use
cols=None, unpack=False, ndmin=0, encoding='bytes')
```

This tells us that `loadtxt` has one parameter called `fname` that doesn't have a default value, and eight others that do. If we call the function like this:

Python

```
numpy.loadtxt('inflammation-01.csv', ',')
```

then the filename is assigned to `fname` (which is what we want), but the delimiter string `','` is assigned to `dtype` rather than `delimiter`, because `dtype` is the second parameter in the list. However `','` isn't a known `dtype` so our code produced an error message when we tried to run it. When we call `loadtxt` we don't have to provide `fname=` for the filename because it's the first item in the list, but if we want the `','` to be assigned to the variable `delimiter`, we *do* have to provide `delimiter=` for the second parameter since `delimiter` is not the second parameter in the list.

Readable functions

Consider these two functions:

Python

```
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))

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!

Combining Strings

"Adding" two strings produces their concatenation: `'a' + 'b'` is `'ab'`. Write a function called `fence` that takes two parameters called `original` and `wrapper` and returns a new string that has the wrapper character at the beginning and end of the original. A call to your function should look like this:

Python

```
print(fence('name', '*'))
```

Output

```
*name*
```

Solution

Return versus print

Note that `return` and `print` are not interchangeable. `print` is a Python function that *prints* data to the screen. It enables us, *users*, see the data. `return` statement, on the other hand, makes data visible to the program. Let's have a look at the following function:

Python

```
def add(a, b):
    print(a + b)
```

Question: What will we see if we execute the following commands?

Python

```
A = add(7, 3)
print(A)
```

Solution

Selecting Characters From Strings

If the variable `s` refers to a string, then `s[0]` is the string's first character and `s[-1]` is its last. Write a function called `outer` that returns a string made up of just the first and last characters of its input. A call to your function should look like this:

Python

```
print(outer('helium'))
```

Output

```
hm
```

Solution

Rescaling an Array

Write a function `rescale` that takes an array as input and returns a corresponding array of values scaled to lie in the range 0.0 to 1.0. (Hint: If `L` and `H` are the lowest and highest values in the original array, then the replacement for a value `v` should be `(v-L) / (H-L)`.)

Solution

Testing and Documenting Your Function

Run the commands `help(numpy.arange)` and `help(numpy.linspace)` to see how to use these functions to generate regularly-spaced values, then use those values to test your `rescale` function. Once you've successfully tested your function, add a docstring that explains what it does.

Solution

Defining Defaults

Rewrite the `rescale` function so that it scales data to lie between `0.0` and `1.0` by default, but will allow the caller to specify lower and upper bounds if they want. Compare your implementation to your neighbor's: do the two functions always behave the same way?

Solution

Variables Inside and Outside Functions

What does the following piece of code display when run — and why?

Python

```
f = 0
k = 0

def f2k(f):
    k = ((f - 32) * (5.0 / 9.0)) + 273.15
    return k

print(f2k(8))
print(f2k(41))
print(f2k(32))

print(k)
```

Solution

Mixing Default and Non-Default Parameters

Given the following code:

Python

```
def numbers(one, two=2, three, four=4):
    n = str(one) + str(two) + str(three) + str(four)
    return n

print(numbers(1, three=3))
```

what do you expect will be printed? What is actually printed? What rule do you think Python is following?

1. `1234`
2. `one2three4`
3. `1239`
4. `SyntaxError`

Given that, what does the following piece of code display when run?

Python

```
def func(a, b=3, c=6):
    print('a: ', a, 'b: ', b, 'c:', c)

func(-1, 2)
```

1. `a: b: 3 c: 6`
2. `a: -1 b: 3 c: 6`
3. `a: -1 b: 2 c: 6`
4. `a: b: -1 c: 2`

Solution

Readable Code

Revise a function you wrote for one of the previous exercises to try to make the code more readable. Then, collaborate with one of your neighbors to critique each other's functions and discuss how your function implementations could be further improved to make them more readable.

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)`.
- Numbers are stored as integers or floating-point numbers.
- Variables defined within a function can only be seen and used within the body of the function.
- Variables created outside of any function are called global variables.
- Within a function, we can access global variables.
- Variables created within a function override global variables if their names match.
- Use `help(thing)` to view help for something.
- Put docstrings in functions to provide help for that function.
- Specify default values for parameters when defining a function using `name=value` in the parameter list.
- Parameters can be passed by matching based on name, by position, or by omitting them (in which case the default value is used).
- Put code whose parameters change frequently in a function, then call it with different parameter values to customize its behavior.

‹ (/python-novice-inflammation/07-cond/index.html)

› (/pyth novice inflam errors