Programming with Python

(index.html)

Creating Functions

✹ Learning 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 repetative, 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:

```
def fahr_to_kelvin(temp):
    return ((temp - 32) * (5/9)) + 273.15
```

The function definition opens with the word `def`, which is followed by the name of the function and a parenthesized list of parameter names. The body (reference.html#function-body) of the function — the statements that are executed when it runs — is indented below the definition line, typically by four spaces.

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. Calling our own function is no different from calling any other function:

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

```
freezing point of water: 273.15
boiling point of water: 373.15
```

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

Integer division

We are using Python 3, where division always returns a floating point number:

```
$ python3 -c "print(5/9)"
```

```
0.5555555555555556
```

Unfortunately, this wasn’t the case in Python 2:

```
5/9
```

```
0
```

If you are using Python 2 and want to keep the fractional part of division you need to convert one or the other number to floating point:

```
float(5)/9
```

```
0.555555555556
```

```
5/float(9)
```

```
0.555555555556
```

```
5.0/9
```

```
0.555555555556
```

```
5/9.0
```

```
0.555555555556
```

And if you want an integer result from division in Python 3, use a double-slash: ~ {.python} 4//2 {.output} 2 {.python} 3//2 {.output} 1 ~

Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

```
def kelvin_to_celsius(temp):
    return temp - 273.15

print('absolute zero in Celsius:', kelvin_to_celsius(0.0))
```

```
absolute zero in Celsius: -273.15
```

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can compose (reference.html#function-composition) the two functions we have already created:

```
def fahr_to_celsius(temp):
    temp_k = fahr_to_kelvin(temp)
    result = kelvin_to_celsius(temp_k)
    return result

print('freezing point of water in Celsius:', fahr_to_celsius(32.0))
```

```
freezing point of water in Celsius: 0.0
```

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-large 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.

Tidying up

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

```
def analyze(filename):

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

    fig = plt.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(data.mean(axis=0))

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

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

    fig.tight_layout()
    plt.show(fig)
```

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

```
def detect_problems(filename):

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

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

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:

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

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 center a dataset around a particular value:

```
def center(data, desired):
    return (data - data.mean()) + desired
```

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 center that around 3:

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

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

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

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

```
[[-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 simple tests that will reassure us:

```
print('original min, mean, and max are:', data.min(), data.mean(), data.max())
centered = center(data, 0)
print('min, mean, and and max of centered data are:', centered.min(), centered.mean(), c
entered.max())
```

```
original min, mean, and max are: 0.0 6.14875 20.0
min, mean, and and max of centered 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 how about -6.1. The mean of the centered 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:

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

```
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:

```
print('difference in standard deviations before and after:', data.std() - centered.st
d())
```

```
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:

```
# center(data, desired): return a new array containing the original data centered around
the desired value.
def center(data, desired):
    return (data - data.mean()) + desired
```

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:

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired valu
e.'''
    return (data - data.mean()) + desired
```

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

```
help(center)
```

```
Help on function center in module __main__:

center(data, desired)
    Return a new array containing the original data centered around 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:

```
def center(data, desired):
    '''Return a new array containing the original data centered around the desired valu
e.
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''
    return (data - data.mean()) + desired

help(center)
```

```
Help on function center in module __main__:

center(data, desired)
    Return a new array containing the original data centered around the desired value.
    Example: center([1, 2, 3], 0) => [-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=` :

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

```
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=` :

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-e3bc6cf4fd6a> in <module>()
----> 1 numpy.loadtxt('inflammation-01.csv', ',')

/Users/gwilson/anaconda/lib/python2.7/site-packages/numpy/lib/npyio.pyc in loadtxt(fnam
e, dtype, comments, delimiter, converters, skiprows, usecols, unpack, ndmin)
    775     try:
    776         # Make sure we're dealing with a proper dtype
--> 777         dtype = np.dtype(dtype)
    778         defconv = _getconv(dtype)
    779

TypeError: data type "," not understood
```

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

```
def center(data, desired=0.0):
    '''Return a new array containing the original data centered around the desired value
(0 by default).
    Example: center([1, 2, 3], 0) => [-1, 0, 1]'''
    return (data - data.mean()) + desired
```

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

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

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

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

```
more_data = 5 + numpy.zeros((2, 2))
print('data before centering:')
print(more_data)
print('centered data:')
print(center(more_data))
```

```
data before centering:
[[ 5.  5.]
 [ 5.  5.]]
centered 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:

```
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)
```

```
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:

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

```
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` :

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

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

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, ski
prows=0, usecols=None, unpack=False, ndmin=0)
    Load data from a text file.

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

    Parameters
    ----------
    fname : file or str
        File, filename, or generator to read.  If the filename extension is
        ``.gz`` or ``.bz2``, the file is first decompressed. Note that
        generators should return byte strings for Python 3k.
    dtype : data-type, optional
        Data-type of the resulting array; default: float.  If this is a
        record data-type, the resulting array will be 1-dimensional, and
        each row will be interpreted as an element of the array.  In this
        case, the number of columns used must match the number of fields in
        the data-type.
    comments : str, optional
        The character used to indicate the start of a comment;
        default: '#'.
    delimiter : str, optional
        The string used to separate values.  By default, this is any
        whitespace.
    converters : dict, optional
        A dictionary mapping column number to a function that will convert
        that column to a float.  E.g., if column 0 is a date string:
        ``converters = {0: datestr2num}``.  Converters can also be used to
        provide a default value for missing data (but see also `genfromtxt`):
        ``converters = {3: lambda s: float(s.strip() or 0)}``.  Default: None.
    skiprows : int, optional
        Skip the first `skiprows` lines; default: 0.
    usecols : sequence, optional
        Which columns to read, with 0 being the first.  For example,
        ``usecols = (1,4,5)`` will extract the 2nd, 5th and 6th columns.
        The default, None, results in all columns being read.
    unpack : bool, optional
        If True, the returned array is transposed, so that arguments may be
        unpacked using ``x, y, z = loadtxt(...)``.  When used with a record
        data-type, arrays are returned for each field.  Default is False.
    ndmin : int, optional
        The returned array will have at least `ndmin` dimensions.
        Otherwise mono-dimensional axes will be squeezed.
        Legal values: 0 (default), 1 or 2.
        .. versionadded:: 1.6.0

    Returns
    -------
    out : ndarray
        Data read from the text file.
    See Also
    --------
    load, fromstring, fromregex
    genfromtxt : Load data with missing values handled as specified.
    scipy.io.loadmat : reads MATLAB data files

    Notes
    -----
    This function aims to be a fast reader for simply formatted files.  The
    `genfromtxt` function provides more sophisticated handling of, e.g.,
    lines with missing values.

    Examples
    --------
    >>> from StringIO import StringIO   # StringIO behaves like a file object
    >>> c = StringIO("0 1\n2 3")
    >>> np.loadtxt(c)
    array([[ 0.,  1.],
           [ 2.,  3.]])

    >>> d = StringIO("M 21 72\nF 35 58")
    >>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
    ...                      'formats': ('S1', 'i4', 'f4')})
    array([('M', 21, 72.0), ('F', 35, 58.0)],
          dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])

    >>> c = StringIO("1,0,2\n3,0,4")
    >>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
    >>> x
    array([ 1.,  3.])
    >>> y
    array([ 2.,  4.])
```

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

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skip
rows=0, usecols=None,
        unpack=False, ndmin=0)
```

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:

```
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.

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:

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

```
*name*
```

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:

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

```
hm
```

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 $(v - L)/(H - L)$.)

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.

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?

Variables inside and outside functions

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

```
f = 0
k = 0

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

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

print(k)
```