Writing your own functions

Lesson 6 – 9/23/16

With notes!

Today's schedule

1. Defining your own functions
   - basics
   - importing from a separate file
   - variable "scope"

Defining your own functions

Why do it?

- Allows you to re-use a certain piece of code without re-writing it
- Organizes your code into functional pieces
- Makes your code easier to read and understand

Defining a function

Syntax:

```
def function_name(parameters):
    statements
    var = something
    return var
```

Example:

```
def strAdd(num1, num2):
    result = int(num1) + int(num2)
    return result
```

Example: this is a silly example of a function that can add two numbers together when they are in string form.
Function names follow the same rules as variable names, pretty much.

Defining a function

Syntax:

```
def function_name(parameters):
    statements
    var = something
    return var
```

This is the value that the function returns when we use it. To give a familiar example, the `int()` function's return value is the string converted to an integer.

Which value we return must be considered carefully, since no other information inside the function will be accessible when we call it. All we can do is capture the return value.

Example:

```
def strAdd(num1, num2):
    result = int(num1) + int(num2)
    return result
```

Example: this is a silly example of a function that can add two numbers together when they are in string form.

Using a custom function

```
def strAdd(num1, num2):
   result = int(num1) + int(num2)
   return result
```

Using a custom function

```
def strAdd(num1, num2):
   result = int(num1) + int(num2)
   return result

first = raw_input("First number? ")
second = raw_input("Second number? ")
added = strAdd(first, second)
print added
```

Using a custom function

```
def strAdd(num1, num2):
   result = int(num1) + int(num2)
   return result

first = raw_input("First number? ")
second = raw_input("Second number? ")
added = strAdd(first, second)
print added
```

Function must be defined before it can be used (usually we define all our definitions at the very top of the script or in a separate script)

Here is where execution actually starts (the first un-indented line)

Here is where we "call" our function

Using a custom function

```
4   def strAdd(num1, num2):
5      result = int(num1) + int(num2)
6      return result

1   first = raw_input("First number? ")
2   second = raw_input("Second number? ")
3/7 added = strAdd(first, second)
8   print added
```

Using a custom function

START

```
4   def strAdd(num1, num2):
5      result = int(num1) + int(num2)
6      return result

1   first = raw_input("First number? ")
2   second = raw_input("Second number? ")
3/7 added = strAdd(first, second)
8   print added
```

When python starts a script that has function definitions at the top, it skips those definitions entirely. It will only use them if they are called from somewhere in the main script body. Python looks for the first un-indented line to determine where it should start executing.

Using a custom function

START

```
4   def strAdd(num1, num2):
5      result = int(num1) + int(num2)
6      return result

1   first = raw_input("First number? ")
2   second = raw_input("Second number? ")
3/7 added = strAdd(first, second)
8   print added
```

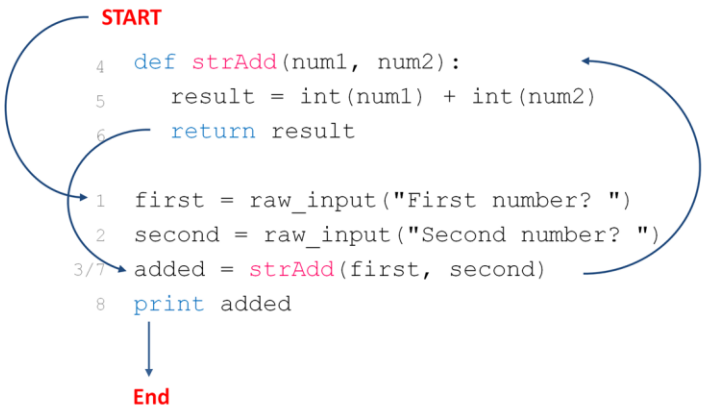
Using a custom function

START

```
4   def strAdd(num1, num2):
5      result = int(num1) + int(num2)
6      return result

1   first = raw_input("First number? ")
2   second = raw_input("Second number? ")
3/7 added = strAdd(first, second)
8   print added
```

Using a custom function



What will this code print?

```
def strAdd(num1, num2):
   result = int(num1) + int(num2)
   return result

first = raw_input("First number? ")
second = raw_input("Second number? ")
added = strAdd(first, second)
print added
```

Result:

```
First number? <input> 5
Second number? <input> 4
```

Assuming we input these values for `first` and `second`

What will this code print?

```
def strAdd(num1, num2):
   result = int(num1) + int(num2)
   return result

first = raw_input("First number? ")
second = raw_input("Second number? ")
added = strAdd(first, second)
print added
```

Result:

```
First number? <input> 5
Second number? <input> 4
9
```

So we used raw input to get numbers (in the form of strings) and then called a single function to both convert them to ints and to add them
If adding two int-strs together was something you had to do a lot, maybe this would be a function worth making (probably not though, since it doesn't save you much typing. A better function might be a wrapper for raw_input() that auto-converts integers when they're they're entered..)

A more useful example: counting

Result of using `.count()`:

```
>>> seq = "CGCACGCACGCGC"
>>> seq.count("CGC")
3
```

Notice that there are actually 4 possible instances of "CGC" in this sequence – the "CGCGC" at the end can be counted as having two instances.

The `.count()` only counts non overlapping instances. What if that's not what we want?

A more useful example: counting

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
    seq = seq.upper()
    subseq = subseq.upper()
    count = 0
    index = 0
    done = False
    while not done:
        index = seq.find(subseq, index)
        if (index == -1):
            done = True
        else:
            count += 1
            index += 1 # add one so this pos won't be found again
    return count

# main script
seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = count_occurrences(seq, subseq)
print "The subseq occurs", result, "times in the full seq"
```

Since this is something that may occur often, we can put our code in a function so that we can use it multiple times in our code without having to copy and paste it.

A more useful example: counting

Result of using `.count()`:

```
>>> seq = "CGCACGCACGCGC"
>>> seq.count("CGC")
3
```

Result:

```
Full sequence: CGCACGCACGCGC
Subseq to search for: CGC
The subseq occurs 4 times in the full seq
```

Keep your functions in a separate file

If you have a set of functions you want to use in various different scripts (e.g. a function to read in a fasta file), you can save these functions in a separate file and then *import* them into other scripts. Example:

useful_fns.py:

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
    seq = seq.upper()
    subseq = subseq.upper()
    count = 0
    index = 0
    done = False
    while not done:
        index = seq.find(subseq, index)
        if (index == -1):
            done = True
        else:
            count += 1
            index += 1
    return count
```

test.py:

```
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print "The subseq occurs", result, "times"
```

Result:

```
> python test.py
Full sequence: CGCACGCACGCGC
Subseq to search for: CGC
The subseq occurs 4 times
```

To make this function maximally useful, we can keep it in a separate file
That way if we ever need to change it (e.g. we find a bug), we only need to change it once, and all other scripts that use it will automatically be up to date
If, on the other hand, we just copied and pasted this code into each script, we'd have to go through and fix every instance. This can be very annoying, and can also cause more bugs.
Note, if we want to use one piece of code that works for many situations, we have to make it as generic as possible. That is, we want to write it in such a way that it will work for pretty much any situation we can imagine.

Keep your functions in a separate file

If you have a set of functions you want to use in various different scripts (e.g. a function to read in a fasta file), you can save these functions in a separate file and then *import* them into other scripts. Example:

useful_fns.py:

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
    seq = seq.upper()
    subseq = subseq.upper()
    count = 0
    index = 0
    done = False
    while not done:
        index = seq.find(subseq, index)
        if (index == -1):
            done = True
        else:
            count += 1
            index += 1
    return count
```

test.py:

```
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print "The subseq occurs", result, "times"
```

we save the file of functions as useful_fns.py, but then import it using just the file name (no .py). Then we can access the functions in this file by saying `useful_fns.functionName()`

Result:

```
> python test.py
Full sequence: CGCACGCACGCGC
Subseq to search for: CGC
The subseq occurs 4 times
```

To make this function maximally useful, we can keep it in a separate file
That way if we ever need to change it (e.g. we find a bug), we only need to change it once, and all other scripts that use it will automatically be up to date
If, on the other hand, we just copied and pasted this code into each script, we'd have to go through and fix every instance. This can be very annoying, and can also cause more bugs.
Note, if we want to use one piece of code that works for many situations, we have to make it as generic as possible. That is, we want to write it in such a way that it will work for pretty much any situation we can imagine.

A note on "scope"

- Variables you *create* within a function are considered to be in a different "scope" than the rest of your code
- This means that those variables are inaccessible outside of the function definition block
- Reusing a variable name within a function definition block will not overwrite any variable defined outside the block.
- Somewhat confusingly, functions *can* sometimes use variables defined within the main body (as long as it has been created before the function is called). However, doing this generally considered bad practice, since it makes the effects of a function harder to predict (especially if you plan to use it in many different scripts).
- The best practice is to only allow functions to use the external variables that are supplied directly as parameters.

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)
```

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)
```

function scope

main scope

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)

>>> print result
```

function scope

main scope

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)

>>> print result
2500
```

function scope

main scope

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)

>>> print result
2500
>>> print z
```

function scope

main scope

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)
```

function scope

main scope

```
>>> print result
2500
>>> print z
1
```

The z defined in the main scope was not overwritten by the z defined in the function scope

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)
```

function scope

main scope

```
>>> print result
2500
>>> print z
1
>>> print c
```

Example of scope

```
>>> def someFn(val):
...     c = val * 10
...     z = c * c
...     return z
...
>>> x = 5
>>> z = 1
>>> result = someFn(x)
```

function scope

main scope

```
>>> print result
2500
>>> print z
1
>>> print c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

There is no `c` defined in the main scope, and we cannot access the `c` defined in the function scope, so this creates a `NameError`