Writing modular code: Functions & command line args

Programming Bootcamp 2015

Day 6 – 6/19/15

Sign in!

Today's schedule

1. Defining your own functions
2. Using command line arguments
3. Good programming practices

1. Defining your own functions

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
```

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)

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

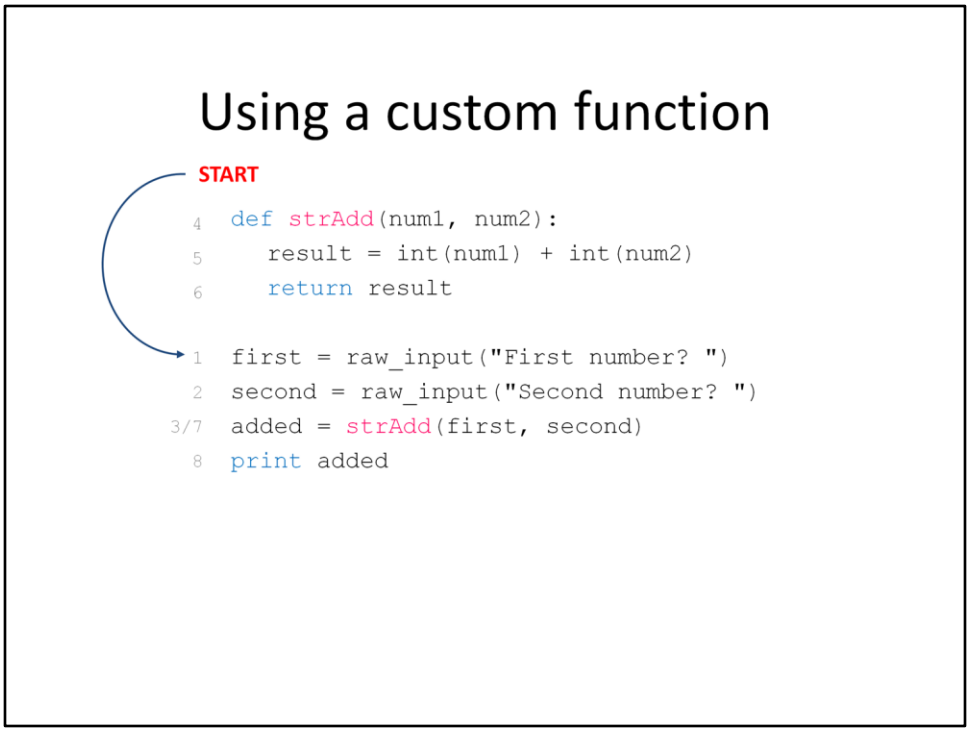
Here is where execution actually starts (the first un-indented line)

Here is where we "call" our function

Using a custom function

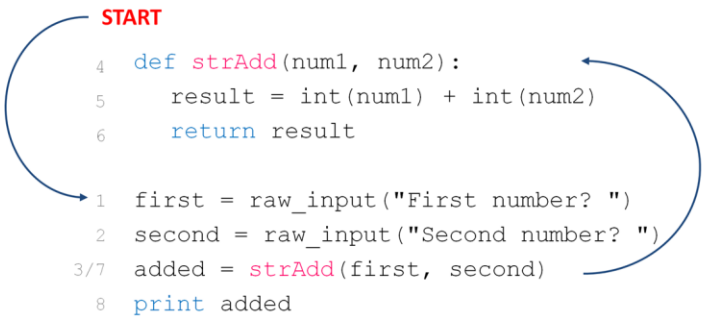
```
4  def strAdd(num1, num2):
5      result = int(num1) + int(num2)
6      return result

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

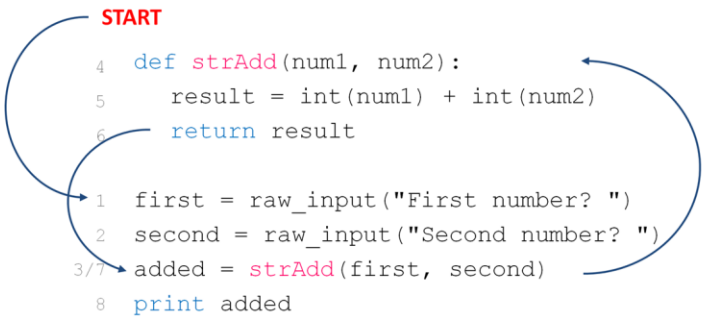


When python starts a script that has function definitions at the top, it skips those definitions entirely. It will only use them if they are called from somewhere in the main script body. Python looks for the first un-indented line to determine where it should start executing.

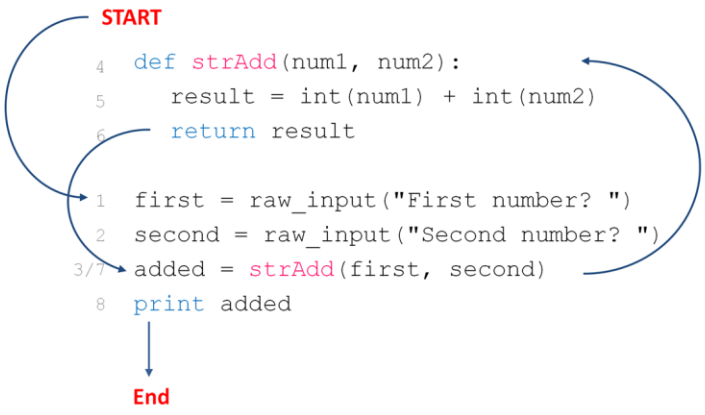
Using a custom function



Using a custom function



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:

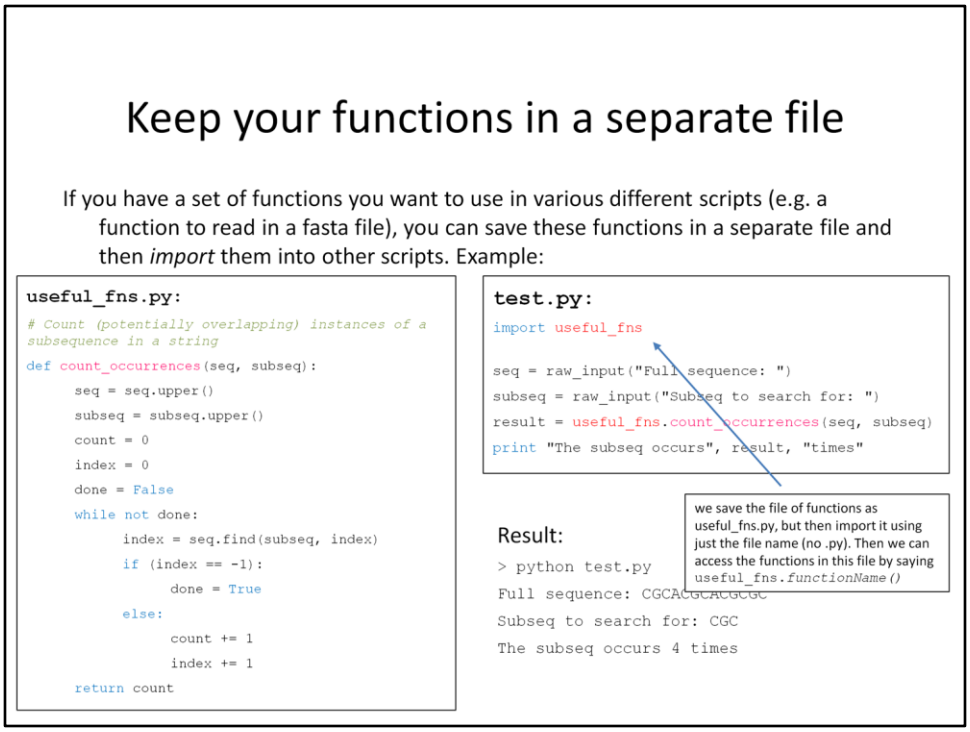
```
import useful_fns

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.



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)

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

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
>>> 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`

2. Command line args

Command line arguments

Usually when we run a python script, we type this into the terminal:

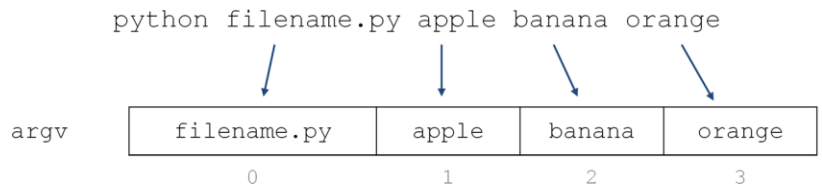
```
python filename.py
```

We can also provide additional information when we run our script ("arguments"):

```
python filename.py arg1 arg2 agr3
```

Command line arguments

You can add as many command line args as you want. All args will be automatically stored (in order) in a list called `argv`. The first item in this list will be the name of your script, followed by any arguments you included.



You do not need to create argv, it is automatically created every time you run a script. Even if you don't have any args, this list still holds the name of the script that was called, so you can use this to get that info if for some reason you need it.

Using `argv`

Before we can use the `argv` list, we must import the `sys` package. This package is already included with your installation of python, we just have to tell python to import it when we start our script.

To do this, simply put this line at the top of your script:

```
import sys
```

We then access argv by typing:

```
sys.argv[...]
```

Example: `argTest.py`

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print "Script", scriptName, "had args:", arg1, arg2
```

Result

```
> python argTest.py apple banana
Script argTest.py had args: apple banana
```

Example: `argTest.py`

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print "Script", scriptName, "had args:", arg1, arg2
```

What if we did this? (only one arg provided)

```
> python argTest.py apple
```

Example: `argTest.py`

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print "Script", scriptName, "had args:", arg1, arg2
```

What if we did this? (only one arg provided)

```
> python argTest.py apple
Traceback (most recent call last):
  File "argTest.py", line 5, in <module>
    arg2 = sys.argv[2]
IndexError: list index out of range
```

Example 2: `addMe.py`

To gracefully exit when the wrong arguments are provided, you could do something like this:

```
import sys

if len(sys.argv) == 3:
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
else:
    print "You must provide two numbers. Exiting."
    sys.exit()

print num1 + num2
```

Result

```
> python addMe.py 100 50
150
```

Or:

```
> python addMe.py 302
You must provide two numbers. Exiting.
```

Example 2: `addMe.py`

To gracefully exit when the wrong arguments are provided, you could do something like this:

```
import sys

if len(sys.argv) == 3:
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
else:
    print "You must provide two numbers. Exiting."
    sys.exit()

print num1 + num2
```

Check if the length of the `argv` list is what we expect. *Remember the script name is the first arg, so a script with 2 args has an `argv` of length 3.

If not, use this piece of code to immediately terminate the whole script.

Result

```
> python addMe.py 100 50
150
```

Or:

```
> python addMe.py 302
You must provide two numbers. Exiting.
```

Why use command line args?

- If you plan to run your script on multiple datasets, you can simply supply different filenames to the command instead of editing a hard-coded file name
- Facilitates the creation of "pipelines", for the above reason
- If you are keeping track of what commands you run on your data (which you should!), having all the relevant info as part of the command itself (the file name, certain parameters, etc.) makes what you did more transparent and reproducible.
- The rule of thumb is: if you NEVER plan to change a variable, no matter what dataset you run your code on, it's ok to hard code it. Otherwise, consider making it a command line arg.
- Later we'll go over how to make more user-friendly command line args (e.g. `-h --infile=file.txt --verbose`)

3. Coding best practices

Some guidelines

- Writing code that is **clear** is more important than writing code that is **concise**
  - so doing something in two steps instead of one is totally fine if it makes your code clearer!
- Comment your code
  - avoid "obvious" comments
  - make sure to keep comments accurate if the code changes
- Use descriptive variable names
  - this goes hand in hand with clarity
- Test your code using small test sets
  - especially important for research! small errors can lead to big mistakes...
- Avoid copy-pasting code -- if you use the same code multiple times, consider making it a function!

The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Notice there's nothing about efficiency here. In general, at least in the Python world, clarity and readability is valued more highly than pure efficiency (although you should always do both, if you can!)

The bolded ones are the ones I think are most important.

Some additional opinions

- https://www.python.org/dev/peps/pep-0008/
- http://docs.python-guide.org/en/latest/writing/style/
- http://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118
- many more...

Note: don't worry if you don't do everything in these guidelines (you'll notice I don't do many of them!). Some points are just more important than others, and you should try to understand why something is a guideline before you blindly follow it 100% of the time.

For more discussion of this:

http://programmers.stackexchange.com/questions/14856/what-popular-best-practices-are-not-always-best-and-why