



Writing your own functions

Lesson 4: 9/21/2017

Speaker: Samantha Klasfeld

Today's Schedule

- Defining your own functions
 - basics
 - importing from a separate file
 - variable “scope”

1. Basics

Defining you 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
```

Function names follow the same rules as variable names, pretty much.

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
```

Using a custom function

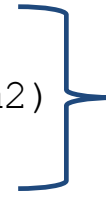
```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result
```


Using a custom function

```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result  
  
first = input("First number? ")  
second = 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 = input("First number? ")  
second = 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 = input("First number? ")  
second = input("Second number? ")  
added = strAdd(first, second)  
print(added)
```

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

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 = input("First number? ")  
second = 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

- 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

- 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.
- For Example:

```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result  
  
first = input("First number? ")  
second = input("Second number? ")  
added = strAdd(first, second)  
print(added)
```

Using a custom function

- 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.
- For Example:

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

START HERE
INSTEAD → `first = input("First number? ")`
 `second = input("Second number? ")`
 `added = strAdd(first, second)`
 `print(added)`

Using a custom function

- 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.
- For Example:

execution order

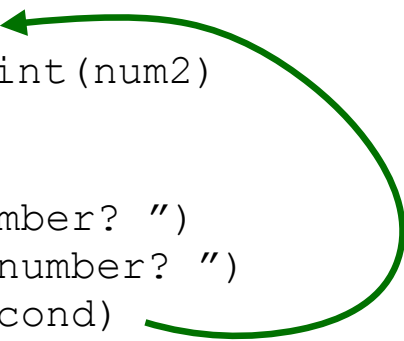
```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result  
  
1    first = input("First number? ")  
2    second = input("Second number? ")  
3    added = strAdd(first, second)  
    print(added)
```


Using a custom function

- 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.
- For Example:

execution order

```
4      def strAdd(num1, num2):  
5          result = int(num1) + int(num2)  
6          return result  
  
1      first = input("First number? ")  
2      second = input("Second number? ")  
3      added = strAdd(first, second)  
      print(added)
```

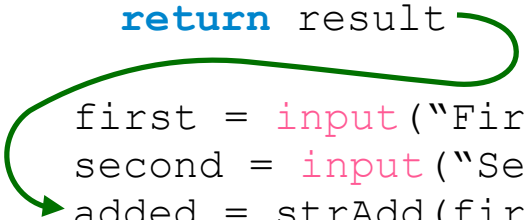


Using a custom function

- 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.
- For Example:

execution order

```
4      def strAdd(num1, num2):  
5          result = int(num1) + int(num2)  
6          return result  
  
1      first = input("First number? ")  
2      second = input("Second number? ")  
3,7    added = strAdd(first,second)  
8      print(added)
```



Using a custom function

- 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.
- For Example:

execution order

```
4      def strAdd(num1, num2):  
5          result = int(num1) + int(num2)  
6          return result  
  
1      first = input("First number? ")  
2      second = input("Second number? ")  
3,7    added = strAdd(first,second)  
8      print(added)
```

↓
END

What will this code print?

```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result  
  
first = input("First number? ")  
second = input("Second number? ")  
added = strAdd(first, second)  
print(added)
```

Result:

First number? <input> 5	}	<div>Assuming we input these values for first and second</div>
Second number? <input> 4		

What will this code print?

```
def strAdd(num1, num2):  
    result = int(num1) + int(num2)  
    return result  
  
first = input("First number? ")  
second = input("Second number? ")  
added = strAdd(first, second)  
print(added)
```

Result:

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

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 = input("Full sequence: ")
subseq = 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.

`find` returns index if found and -1 otherwise.

A more useful example: counting

Result of using `.count()`

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

Result of using running script:

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


2. Importing from a separate file

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 = input("Full sequence: ")
subseq = 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
```

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 them in a separate file and then *import* them into other scripts.

we save the file of functions as `useful_fns.py`, but then import it using just the file name (no `.py`).

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 = input("Full sequence: ")
subseq = 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
```

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 them in a separate file and then *import* them into other scripts.

we save the file of functions as `useful_fns.py`, but then import it using just the file name (no `.py`).

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

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
```

Why keep functions in a separate file?

- 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 we just copied and pasted this code into each script, we'd have to go through and fix every instance
- 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.

```
# non-generic functions
def multiply_three(int_var):
    return int_var * 3

def multiply_two(int_var):
    return int_var * 2

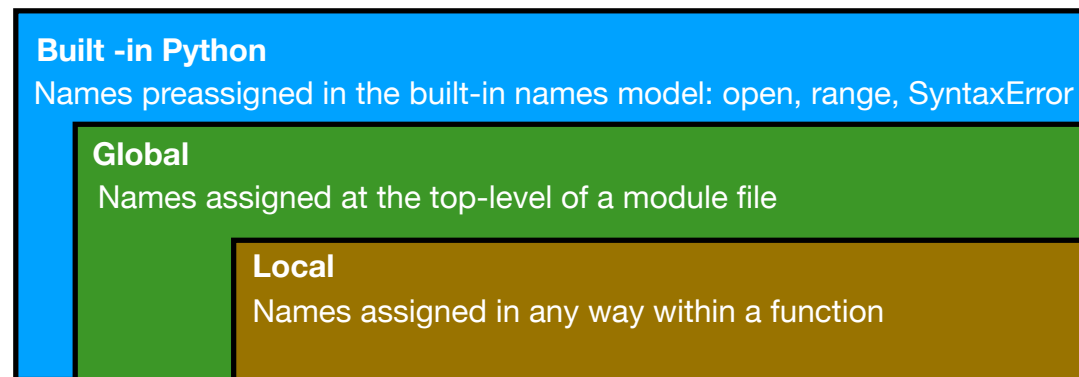
# improved generic function
def multiply(int_var1, int_var2):
    return int_var1 * int_var2
```

- more code to fix means more likely to accidentally leave bugs
- more likely to accidentally leave bugs means more likely to have a frustrated programmer

3. variable “scope”

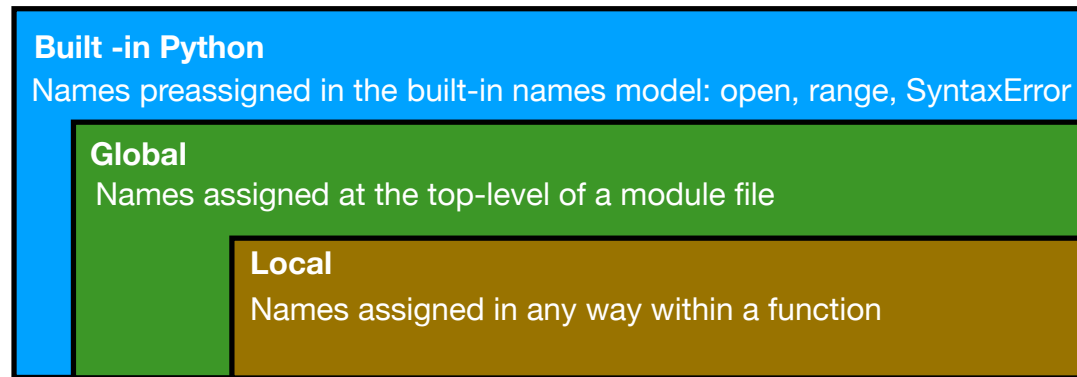
What is a variable's “scope”?

- The scope of a variable is the location of a name's assignment in your source code
- the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility
- For this class we can focus on three scopes



Note that there is a `global()` function that can allow objects to reach out of their scope, but we will not be discussing that...

What is a variable's "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 local definition block
 - Reusing a variable name within a function definition block will not overwrite any variable defined outside the block.

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)
```

} local scope

} global scope

Example of scope

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

} local scope

} global 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
```

} local scope

} global 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)
```

} local scope

} global 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
```

} local scope

} global scope

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

Example of scope

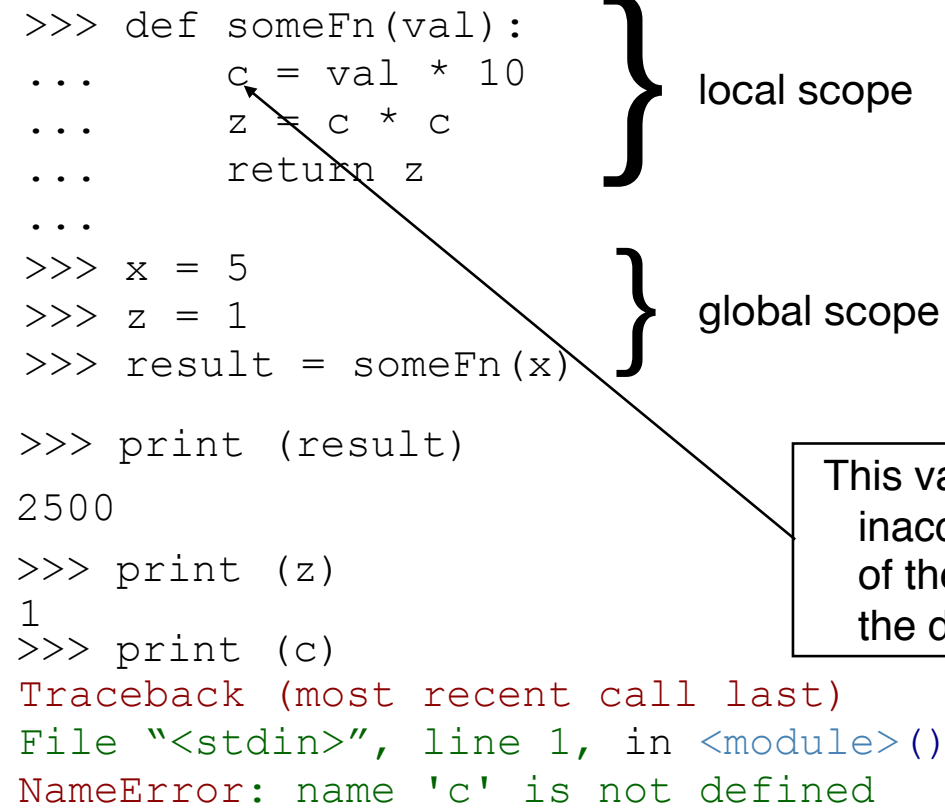
```
>>> def someFn(val):  
...     c = val * 10  
...     z = c * c  
...     return z  
...  
>>> x = 5  
>>> z = 1  
>>> result = someFn(x)  
  
>>> print (result)  
2500  
>>> print (z)  
1  
>>> print (c)
```

} local scope

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



local scope

global scope

This variables is inaccessible outside of the local scope of the definition

scope WARNING

- 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 bad practice
int_var2 = 3
def multiply_three(int_var1):
    return int_var1 * int_var2

multiply_three(4)
```

```
# example of good practice
def multiply(int_var1, int_var2):
    return int_var1 * int_var2

multiply(4,3)
```

scope WARNING

- 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 bad practice
int_var2 = 3
def multiply_three(int_var1):
    return int_var1 * int_var2

print(multiply_three(4))
```

```
# example of good practice
def multiply(int_var1, int_var2):
    return int_var1 * int_var2

print(multiply(4, 3))
```

Result (in both): 12