# Lecture 7 - Writing functions, data vs references

Kevin Bonham, PhD

2018-09-25

## Outline

- Homework notes
- Functions and how to write them
- Scope and the difference between data and references

## Learning Objectives

After this lecture, you will be able to:

- Write python functions with scalar arguments that return values
- Explain the difference between mutable and immutable objects
- Identify the scope of a variable based on its location in loops and functions

## Homework Notes

- Follow instructions!
- Be sure that your code runs wothout errors!
- For homework 3, you will need to be able to load a file in your code

## Computer programs are data + actions

- Data can be scalars
- Data can be collections
- Functions perform actions

## ”Functions” perform actions on data

```
In [ ]: import time
```

```
In [ ]: print("Hello, World!")

        counter = 0
        for i in range(5):
            counter = counter + 1
            print(counter)
            time.sleep(2)
```

## Defining a function

In python...

- Function definitions start with `def`
- function names can start with _ or a letter (NOT a number)
- function names can contain upper and lowercase letters, numbers and _
  - python convention: use lowercase and _ to separate words
- functions have `(` immediately after name, args, and end with `):` followed by a line break and a tab (or 2+ spaces)

In [ ]:
```
def my_function_name(arg1, arg2, kwarg1="default"):
    # Code that performs actions on args
    return None # this is what is returned by default
```

In [ ]:
```
my_function_name(1,2)
```

## Args are like variables inside a function

In [ ]:
```
x = "a string"
print(x + " woo!")
```

In [ ]:
```
def a_func(y):
    print(y + " woo!")
```

In [ ]:
```
a_func("other string")
a_func("string 3")
a_func("4")
```

## Use it in a loop

```
In [ ]: for i in range(10):
            a_func(str(i))
```

## Anytime you do something more than ~ twice, probably write a function

In [ ]:

```
# last week you saw this:
traffic_signal = "Yellow"
if traffic_signal == "Green":
    print( "Let's go!" )
elif traffic_signal == "Yellow":
    print( "Slow down, prepare to stop." )
elif traffic_signal == "Red":
    print( "Stop!" )
else:
    print( "Unknown signal; proceed with caution." )

# Do you want to write all this any time you see a traffic signal? No!
```

In [ ]:
```python
def read_traffic_signal(a_traffic_signal):
    if a_traffic_signal == "Green":
        print( "Let's go!" )
    elif a_traffic_signal == "Yellow":
        print( "Slow down, prepare to stop." )
    elif a_traffic_signal == "Red":
        print( "Stop!" )
    else:
        print( "Unknown signal; proceed with caution." )
```

In [ ]:
```python
read_traffic_signal("Green")
read_traffic_signal("Yellow")
read_traffic_signal("Blue")
```

## Functions can contain any valid code

- assign variables
- perform actions in loops
- check things with conditionals (`if`/`elif`/`else`)
- call other functions (or even themselves!)
- even define other functions

## Poll 1:

In [ ]:
```
z = 10

def oops_bad_idea(z):
    return z + 5
```

1. Given the code above, what will I get when I run `oops_bad_idea(20)`?
2. After I run `oops_bad_idea(20)`, what is the value of z?

## **args** VS **kwargs**

```
In [ ]: def takes_positional_args(a, b):
            return a**2 + b
```

```
In [ ]: takes_positional_args()
```

## Arguments are taken in order

```
In [ ]: takes_positional_args(1,2)
```

```
In [ ]: takes_positional_args(a=10, b=100)
```

```
In [ ]: # order doesn't matter if providing keywords
        takes_positional_args(b=100, a=10)
```

## You can provide default values

```
In [ ]: def takes_kwargs(m=5, n=20):
            return m + n
```

```
In [ ]: takes_kwargs()
```

```
In [ ]: takes_kwargs(n=500)
```

## Positional arguments can’t be mixed and matched

```
In [ ]: takes_kwargs(10, m=11)
```

## Variables have ”scope”

- Variables defined at the "top level" (eg not in functions or loops) have "global scope"
- other variables have scope limited to the block in which they were defined
- this can cause unintuitive results!

In [ ]:

```
def foo(rand_arg):
    return # this doesn't do anything

foo(5)
print(rand_arg)
```

In [ ]:

```
for i in range(10):
    continue # this doesn't do anything

print(i)
```

In [ ]:

```python
def bar():
    for j in range(10):
        continue

bar()
print(j)
```

In [ ]:

```
def bar():
    for j in range(10):
        continue
    return j

bar()
print(j)
```

In [ ]:
```
print(bar())
```

## Data vs References

- Variables are references to data, not the data itself
- For datatypes that are immutable (most scalar types), this doesn't matter
- For other datatypes (collections, classes), it can matter a lot

# Examples

## Note: you are responsible for the new concepts that are demonstrated in the examples below

These examples are meant to demonstrate the difference between data and references to data. Execute the cells in order, and be sure that you can answer all of the questions in **bold**, and that the answers make sense.

By "make sense," I mean that you can understand the behavior of the code, not that you would have made the same design decisions :-). **NOTE:** Pay attention to the errors that are in my original code. Being able to recognize error types and what they mean can make your life A LOT easier when writing your own code.

You can also edit the code and re-execute to try different variations (I suggest creating new cells and new variable names so as not to mess with my examples - you can also start a new notebook or work in the python REPL to keep things truly separate).

In [ ]:
```
# these are immutable
a_float = 3.1
a_string = "Hello, World!"

# this is mutable
a_list = ["hello {}".format(n) for n in range(3)]
```

```
In [ ]: a_list
```

## Alright - pay attention...

```
In [ ]: another_list = a_list
```

`a_list` and `another_list` now refer to the same underlying object

```
In [ ]: another_list
```

```
In [ ]: another_list == a_list
```

```
In [ ]: a_list is another_list
```

```
In [ ]: a_list.append("Look at me! I'm propagating")
```

### Are `a_list` and `another_list` still the same?

```
In [ ]: a_list == another_list
```

When we evaluated `a_list.append()`, we modified the underlying list object that both variables, `a_list` and `another_list`, are referencing.

```
In [ ]: a_list = ["I", "don't", "like", "change"]
```

**What about now? Are `a_list` and `another_list` still the same?**

```
In [ ]: a_list == another_list
```

```
In [ ]: a_list
```

```
In [ ]: another_list
```

When we evaluated `a_list = ...`, we are **reassigned** the variable `a_list` to refer to a different object.

## `is` vs `==`

Python has a nifty bit of syntax that helps us determine if something is refering to the same object vs those that simply have the same value.

In [ ]:
```
x = 4
y = 4
```

In [ ]:
```
x == y
```

In [ ]:
```
x is y
```

`int`s and `str`s are immutable - there's no difference between having the same value and being the same object. Since you can't change the objects, only reasign the variable referring to them, == and `is` are the same

In [ ]:
```
y = y + 2
```

**Does the assignment above alter the value of `x`?**

In [ ]:
```
x == y
```

In [ ]:
```
print("the value of x is", x)
print("the value of y is", y)
```

We didn't "mutate" 4 (we can't! `int`'s are immutable!). We simple reassigned `y` to a different value.

In [ ]:
```
w = ["a", "list"]
v = ["a", "list"] # this is a different object with the same value
```

## What will == and `is` return for `w` and `v`?

In [ ]:
```
w == v
```

In [ ]:
```
w is v
```

## How are `w` and `v` different from `a_list` and `another_list` above?

In [ ]:
```
a_list = another_list = ["let's", "see", "that", "again"]
```

In [ ]:
```
w.append("not propagating!")
```

In [ ]:
```
v
```

In [ ]:
```
w
```

In [ ]:
```
a_list.append("propagating!")
```

In [ ]:
```
a_list
```

In [ ]:
```
another_list
```

In [ ]:
```
a_list is another_list
```

## Putting it together with scope and functions

Pay attention to the scope of variables, and to whether variables refer to objects that are mutable or not.

In [ ]:
```
from math import sqrt

sqrt(4)
```

In [ ]:
```
sqrt(-4)
```

#### Write a function that returns -1 if value provided is negative, and returns the squareroot if it's zero or positive

```
In [ ]: def safe_sqrt(my_num): # don't change this line
            if my_num ???:
                # What should be done if the conditional is true?
            else:
                # what if it's not?
```

Does your function return something? **Note:** you can have multiple `return` statements in your function. Whichever one is reached first will be evaluated and the function will complete.

```
In [ ]: # If it works, this should return True
        safe_sqrt(-4) == -1
```

```
In [ ]: # If it works, this should return True
        safe_sqrt(9) == 3
```

## Will the following return `True` or `False`?

```
In [ ]: 3 is 3.0
```

```
In [ ]: my_num = -4

        safe_sqrt(16) # -1 or 4?
```

## Assignment and conditionals

Be careful with conditionals that cause variables to be assigned. It's usually a good idea to assign the variable with a default value outside of the conditional, and then modify it inside. Execute the following cells in order, note the outputs / errors.

**Can you explain what's happening?**

In [ ]:
```
x1 = 4

if x1 % 2 == 0: # do you remember what % means?
    y1 = 3
```

In [ ]:
```
y1
```

In [ ]:
```
x2 = 5

if x2 % 2 == 0: # do you remember what % means?
    y2 = 3
```

In [ ]:
```
y2
```

This is especially important in functions, because you don't necessarily know what will be passed as arguments.

In [ ]:
```
def is_even(some_number):
    if some_number % 2 == 0:
        answer = True

    return answer
```

In [ ]:
```
is_even(6)
```

In [ ]:
```
is_even(7)
```

What's a better way to write this function so that it can take any integer and give the correct answer? Bonus points* if you include a way to check for invalid inputs (like negative numbers or floats).

*there are no points for this.

In [ ]:
```
def better_is_even(some_number):
    # your code here
    return answer
```

In [ ]:
```
better_is_even(7) # this should return False
```

In [ ]:
```
better_is_even(-2) # what does this return? What should it return?
```

## Functions that take mutables

Things start to get more complicated when you write functions that take lists and other collections. Watch out!

In [ ]:
```
def append_mean(some_list):
    # this check is not necessary, but it's nice to give your users
    # (by user I mean yourself in 2 weeks) some indication of what went wrong
    if not type(some_list) == list:
        raise ValueError("Hey! this function needs a list")

    list_mean = sum(some_list) / len(some_list)
    some_list.append(list_mean)
    return list_mean
```

What does the function above do? **NOTE:** it actually does **2** things that will appear outside the scope of the function (I'm counting the return value).

In [ ]:
```
test_list = [10, 3, 7]
```

*Evaluate the next cell multiple times*, note the output. Is it consistant?

In [ ]:
```
append_mean(test_list)
```

What's in `test_list`? Is it what you were expecting?

```
In [ ]: test_list
```

## Weird behavior with collections as default arguments

Probably just don't do it

```
In [ ]: def dont_do_this(lst=[]):
            lst.append(1)
            return lst
```

```
In [ ]: dont_do_this(lst = [-2,-1,0]) # all looks fine if we override the default...
```

```
In [ ]: dont_do_this() # if we use the default it seems ok...
```

```
In [ ]: dont_do_this() # wtf?
```

Why is this happening?