In [8]:

```
import numpy as np
import pandas as pd

# These lines set up graphing capabilities.
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import warnings
warnings.simplefilter('ignore', FutureWarning)

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

# 1. Defining functions

Let's write a very simple function that converts a proportion to a percentage by multiplying it by 100. For example, the value of `to_percentage(.5)` should be the number 50 (no percent sign).

A function definition has a few parts.

#### *`def`*

It always starts with `def` (short for **def**ine):

```
def
```

#### *Name*

Next comes the name of the function. Like other names we've defined, it can't start with a number or contain spaces. Let's call our function `to_percentage` :

```
def to_percentage
```

#### *Signature*

Next comes something called the *signature* of the function. This tells Python how many arguments your function should have, and what names you'll use to refer to those arguments in the function's code. A function can have any number of arguments (including 0!).

`to_percentage` should take one argument, and we'll call that argument `proportion` since it should be a proportion.

```
def to_percentage(proportion)
```

If we want our function to take more than one argument, we add a comma between each argument name. Note that if we had zero arguments, we'd still place the parentheses () after that name.

We put a **colon** after the signature to tell Python that the next indented lines are the body of the function. If you're getting a syntax error after defining a function, check to make sure you remembered the colon!

```
def to_percentage(proportion):
```

#### *Documentation*

Functions can do complicated things, so you should write an explanation of what your function does. For small functions, this is less important, but it's a good habit to learn from the start. Conventionally, Python functions are documented by writing an **indented** triple-quoted string:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
```

#### *Body*

Now we start writing code that runs when the function is called. This is called the *body* of the function and every line **must be indented with a tab**. Any lines that are *not* indented and left-aligned with the def statement is considered outside the function.

Some notes about the body of the function:

- We can write code that we would write anywhere else.
- We use the arguments defined in the function signature. We can do this because we assume that when we call the function, values are already assigned to those arguments.
- We generally avoid referencing variables defined *outside* the function. If you would like to reference variables outside of the function, pass them through as arguments!

Now, let's give a name to the number we multiply a proportion by to get a percentage:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
```

### ***`return`***

The special instruction `return` is part of the function's body and tells Python to make the value of the function call equal to whatever comes right after `return`. We want the value of `to_percentage(.5)` to be the proportion .5 times the factor 100, so we write:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
    return proportion * factor
```

`return` only makes sense in the context of a function, and **can never be used outside of a function**. `return` is always the last line of the function because Python stops executing the body of a function once it hits a `return` statement. If a function does not have a return statement, it will not return anything; if you expect a value back from the function, make sure to include a return statement.

*Note:* `return` inside a function tells Python what value the function evaluates to. However, there are other functions, like `print`, that have no `return` value. For example, `print` simply prints a certain value out to the console.

`return` and `print` are **very** different

**Question 1.1.** Define `to_percentage` in the cell below. Call your function to convert the proportion .2 to a percentage. Name that percentage `twenty_percent`.

In [14]:

```
def to_percentage(proportion=0.2):
    """Converts percentage into proportional"""
    factor =100
    percentage=proportion*factor
    return percentage
twenty_percent=to_percentage(0.2)
print(twenty_percent)
```

```
20.0
```

Like you've done with built-in functions in previous labs (max, abs, etc.), you can pass in named values as arguments to your function.

**Question 1.2.** Use to_percentage again to convert the proportion named a_proportion (defined below) to a percentage called a_percentage.

Note: You don't need to define to_percentage again! Like other named values, functions stick around after you define them.

In [15]:

```
a_proportion=2**(.5)/2
a_percentage=to_percentage(a_proportion)
print(a_percentage)
```

```
70.71067811865476
```

Here's something important about functions: the names assigned *within* a function body are only accessible within the function body. Once the function has returned, those names are gone. So even if you created a variable called `factor` and defined `factor = 100` inside of the body of the `to_percentage` function and then called `to_percentage`, `factor` would not have a value assigned to it outside of the body of `to_percentage`:

In [16]:

```
# You should see an error when you run this.  (If you don't, you might
# have defined factor somewhere above.)
factor
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-16-a219be0dab32> in <module>
      1 # You should see an error when you run this.  (If you don't,
you might
      2 # have defined factor somewhere above.)
----> 3 factor

NameError: name 'factor' is not defined
```

As we've seen with built-in functions, functions can also take strings (or arrays, or tables) as arguments, and they can return those things, too.

**Question 1.3.** Define a function called `disemvowel`. It should take a single string as its argument. (You can call that argument whatever you want.) It should return a copy of that string, but with all the characters that are vowels removed. (In English, the vowels are the characters "a", "e", "i", "o", and "u".) You can use as many lines inside of the function to do this as you'd like.

*Hint:* To remove all the "a"s from a string, you can use `a_string.replace("a", "")`. The `.replace` method for strings returns a new string, so you can call `replace` multiple times, one after the other.

In [17]:

```
def disemvowel(a_string):
    a_rem=a_string.replace("a","")
    a_rem=a_rem.replace("e","")
    a_rem=a_rem.replace("i","")
    a_rem=a_rem.replace("o","")
    a_rem=a_rem.replace("u","")
    return a_rem






# An example call to your function.  (It's often helpful to run
# an example call from time to time while you're writing a function,
# to see how it currently works.)
disemvowel("Can you read this without vowels?")
```

Out[17]:

```
'Cn y rd ths wtht vwls?'
```

***Calls on calls on calls***

Just as you write a series of lines to build up a complex computation, it's useful to define a series of small functions that build on each other. Since you can write any code inside a function's body, you can call other functions you've written.

If a function is a like a recipe, defining a function in terms of other functions is like having a recipe for cake telling you to follow another recipe to make the frosting, and another to make the jam filling. This makes the cake recipe shorter and clearer, and it avoids having a bunch of duplicated frosting recipes. It's a foundation of productive programming.

For example, suppose you want to count the number of characters *that aren't vowels* in a piece of text. One way to do that is this to remove all the vowels and count the size of the remaining string.

**Question 1.4.** Write a function called `num_non_vowels`. It should take a string as its argument and return a number. That number should be the number of characters in the argument string that aren't vowels. You should use the `disemvowel` function you wrote above inside of the `num_non_vowels` function.

*Hint:* The function `len` takes a string as its argument and returns the number of characters in it.

In [18]:

```
a_string="Can you read this without vowels?"
def num_non_vowels(a_string):
    """The number of characters in a string, minus the vowels."""
    x=len(disemvowel(a_string))
    return x

# Try calling your function yourself to make sure the output is what
# you expect.
num_non_vowels(a_string)
```

Out[18]:

```
22
```

Functions can also encapsulate code that *displays output* instead of computing a value. For example, if you call `print` inside a function, and then call that function, something will get printed.

The `movies_by_year` dataset in the textbook has information about movie sales in recent years. Suppose you'd like to display the year with the 5th-highest total gross movie sales, printed in a human-readable way. You might do this:

In [35]:

```
pwd
```

Out[35]:

```
'/home/ashly'
```

In [115]:

```
movies_by_year = pd.read_csv("/home/ashly/Downloads/movies_by_year.csv")
type(movies_by_year)
movies_by_year=movies_by_year.sort_values(by='Total Gross',ascending=False)
#print(movies_by_year)
rank = 5
fifth_from_top_movie_year = movies_by_year.iloc[rank-1][0]
print("Year number for #",rank, "ranked movie in terms of total gross sales wa
s:", fifth_from_top_movie_year)
```

```
Year number for # 5 ranked movie in terms of total gross sales was:
2010
```

After writing this, you realize you also wanted to print out the 2nd and 3rd-highest years. Instead of copying your code, you decide to put it in a function. Since the rank varies, you make that an argument to your function.

**Question 1.5.** Write a function called `print_kth_top_movie_year`. It should take a single argument, the rank of the year (like 2, 3, or 5 in the above examples). It should print out a message like the one above.

*Note:* Your function shouldn't have a `return` statement.

In [116]:

```
def print_kth_top_movie_year(k):
    movie_year = movies_by_year.iloc[k-1][0]
    print("Year number for #",k, "ranked movie in terms of total gross sales wa
s:",movie_year)

# Example calls to your function:
print_kth_top_movie_year(2)
print_kth_top_movie_year(3)
```

```
Year number for # 2 ranked movie in terms of total gross sales was:
2013
Year number for # 3 ranked movie in terms of total gross sales was:
2012
```

In [117]:

```
# interact also allows you to pass in an array for a function argument. It will
# then present a dropdown menu of options.
_ = interact(print_kth_top_movie_year, k=np.arange(1, 10))
```

## `print` is not the same as `return`

The `print_kth_top_movie_year(k)` function prints the total gross movie sales for the year that was provided! However, since we did not return any value in this function, we can not use it after we call it. Let's look at an example of another function that prints a value but does not return it.

In [118]:

```
def print_number_five():
    print(5)
```

In [119]:

```
print_number_five()
```

```
5
```

However, if we try to use the output of `print_number_five()` , we see that the value `5` is printed but we get a TypeError when we try to add the number 2 to it!

In [120]:

```
print_number_five_output = print_number_five()
print_number_five_output + 2
```

```
5

--------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-120-8bfda6b3fa47> in <module>
      1 print_number_five_output = print_number_five()
----> 2 print_number_five_output + 2

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

It may seem that `print_number_five()` is returning a value, 5. In reality, it just displays the number 5 to you without giving you the actual value! If your function prints out a value without returning it and you try to use that value, you will run into errors, so be careful!

Try to add a line of code to the `print_number_five` function (after `print(5)` ) so that the code `print_number_five_output + 5` would result in the value `10` , rather than an error.

In [121]:

```
def print_number_five():
    print(5)
    return 5
print_number_five_output = print_number_five()
print_number_five_output + 5
```

```
5
```

Out[121]:

```
10
```

In [ ]:

```

```