Readings for Unit 5

Back to Top

The questions below are due on Friday July 21, 2023; 10:00:00 PM.

Licensing Information

Portions of these readings were modified or copied verbatim from the very nice book *Think Python 2e* by Allen Downey.

PDF of these readings also available to download: 6s090_reading5.pdf

Table of Contents

1) Introduction

As we have learned throughout the course, functions allow us to abstract away the details of a particular computation so that it can be computed multiple times on different inputs. This week's readings will, first, revisit the details of how Python interprets functions with two more examples. In particular, we'll focus on the issue of *scoping* (deciding how and where Python looks up variable names). Then, we'll discuss the "first-class" nature of Python functions. Finally, we'll introduce some snazzy new syntax.

2) More Examples

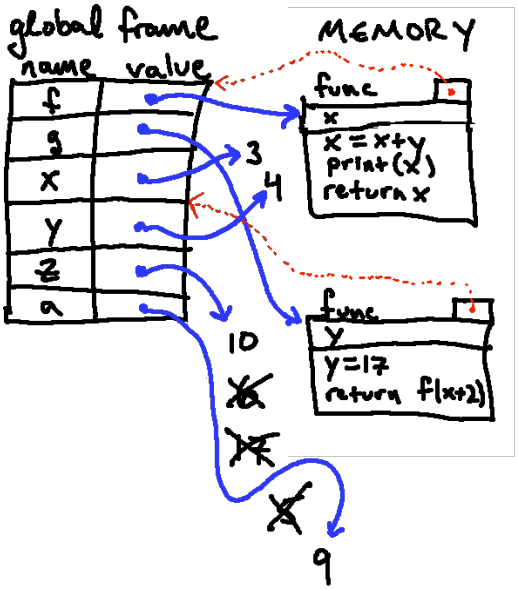
To begin, we will step through two complex function examples with environment diagrams. These both build upon the things we learned in unit 4's reading, so you may wish to review those now.

2.1) Calling Functions From Within Functions, Shadowing Globals

First, let's walk through the following piece of (admittedly silly) code: Back to Top

```
def f(x):
    x = x + y
    print(x)
    return x

def g(y):
    y = 17
    return f(x+2)

x = 3
y = 4
z = f(6)

a = g(y)

print(z)
print(a)
print(x)
print(y)
```

Try to use an environment diagram to predict what values will be printed to the screen as this program runs. You can step through our explanation of how this code runs using the buttons below:



<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 13

This is our final diagram, where the return value from the call to `g` is associated with the name `a` in the global frame.

After executing all of the previous steps, we enter a sequence of print statements. By now, we have printed `10` and `9` already. Now we print `z`, `a`, `x`, and `y`, which have values `10`, `9`, `3`, and `4`, respectively. So all-in-all, we will have printed the following:

Back to Top

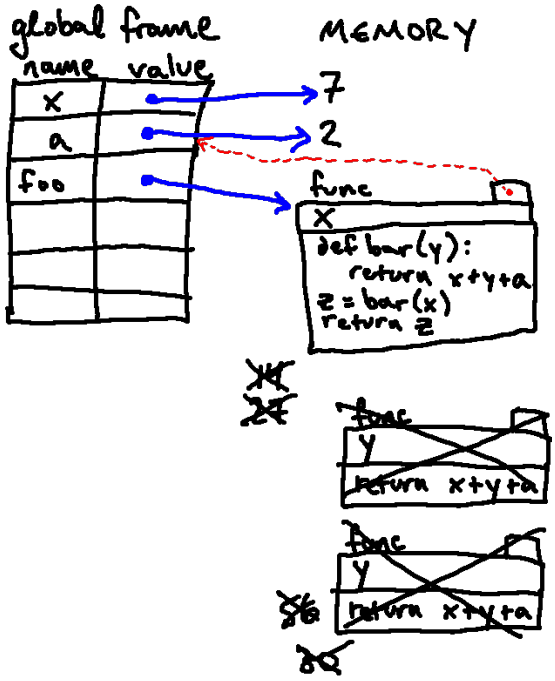
```
10
9
10
9
3
4
```

2.2) Defining a Function Within a Function

As another example, let's walk through the following piece of code. This piece of code demonstrates a new idea: because function bodies can contain arbitrary code, they can also include *other function definitions*! Consider the following code:

```
x = 7
a = 2

def foo(x):
    def bar(y):
        return x + y + a
    z = bar(x)
    return z

print(foo(14))
print(foo(27))
```

Try to use an environment diagram to predict what values will be printed to the screen as this program runs. You can step through our explanation of how this code runs using the buttons below:



Back to Top

[<< First Step] [< Previous Step] [Next Step >] [Last Step >>]

STEP 16

In the process of cleaning up the frame, several values are left with no references to them. These are garbage collected, and then Python exits (because it is finished running the program!)

2.3) A Reminder

If you find these diagrams tedious, we get it... In the end, there's a reason we want computers to be the one doing this, after all; they're much better at these operations than we are, and much faster! So, in the short term, this *is* tedious. But the long-term benefits are really great! This kind of practice is helpful in building up a mental model of Python's behavior, which is important so that when you encounter unexpected behavior, you can come back to the model. With practice, this kind of thinking will become second nature, and you won't have to draw these diagrams out in such detail.

To motivate why environment diagrams might be useful, let's look at another example of a *mystery* Python program:

```
functions = []
for i in range(5):
    def func(x):
        return x + i
    functions.append(func)

for f in functions:
    print(f(12))
```

Without running this code, take a few moments to predict what is going to happen when it is run. Which of the following do you think is going to happen?

Back to Top

(you will not be graded on correctness for this question, just go ahead and mark your best guess)

- ○ It prints `12`, then `13`, then `...`, then `16`
- ○ It prints `13`, then `14`, then `...`, then `17`
- ○ It prints `16`, then `15`, then `...`, then `12`
- ○ It prints `17`, then `16`, then `...`, then `13`
- ○ A Python error occurs
- ○ Something else

Save Submit View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

Now, only once you have made an educated guess above, type this code into your favorite text editor or IDE and run it. Does the result match your expectation? By the end of this reading, we will learn ways to use functions that can 'fix' this program.

3) Functions Are First-Class

We now shift gears to learn about a powerful feature of Python: that it treats functions as first-class objects, which means that functions in Python can be manipulated in many of the same ways that other objects can be (specifically, they can be passed as arguments to other functions, defined inside of other functions, returned from other functions, and assigned to variables). In this section, we will explore how we can make use of this feature in our programs.

3.1) Functions as Arguments

Imagine that you wanted to make plots of several different functions. To do that, you would need to figure out which "y" values correspond to each of a number of "x" values. The following code computes these "y" values for different functions:

```
import math

def sine_response(lo, hi, step):
    out = [] # list of "y" values
    i = lo
    while i <= hi:
        out.append(math.sin(i)) # compute "y" value
        i += step # move to next "x" value
    return out

def cosine_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(math.cos(i))
        i += step
    return out
```

```
def double(x):
    return 2*x

def double_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(double(i))
        i += step
    return out

def square_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(i**2)
        i += step
    return out
```

Back to Top

Now imagine that you wanted the response function to return two lists, one to represent the input values, and one to represent the output values. Making this change or changing anything at all about the functions' behaviors, would be a pain, because you would have to manually change *each* of the above functions. However, we can fix this by making a general function called `response`, which takes a function `f` as input and returns the list of `f`'s outputs over the specified range:

```
def response(f, lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(f(i)) # here, we apply the provided function to i
        i += step
    return out
```

Notice that, inside of the definition of `response`, we call `f`, the function that was passed in as an argument. Using `response`, we could compute the response of our `double` function from earlier:

```
# These two compute the same response!
out = double_reponse(0, 1, 0.1)
out = response(double, 0, 1, 0.1)
```

When we pass in `double` as an argument, we do not put parentheses after it. This is because we want to refer to the function itself (which is called `double`), and not to any particular output of the function (which we'd get by calling it, such as in `double(7)`).

Note that we could compute responses for all of the functions described above using this new `response` function:

```
sine_out = response(math.sin, 0, 1, 0.1)
cosine_out = response(math.cos, 0, 1, 0.1)
double_out = response(double, 0, 1, 0.1)
```

```
def square(x):
    return x**2
square_out = response(square, 0, 1, 0.1)
```

Back to Top

3.2) Function-ception and Returning Functions

Another useful feature is that functions can not only be passed in as arguments to functions, they can also be returned as the result of calling other functions! Imagine that we had the following functions, each designed to add a different number to its input:

```
def add1(x):
    return x+1

def add2(x):
    return x+2
```

If we wanted to make a whole lot of these kinds of functions (`add3`, `add4`, `add5`, ...), it would be nice to have an automated way of making them, rather than defining each new function by hand. We can do this in Python with:

```
def add_n(n):
    def inner(x):
        return x + n
    return inner
```

This may be a little difficult to understand at first, but what is happening is this: when `add_n` is called, it will make a *new function* (here, called `inner`) using the `def` keyword, and it will then return this function.

Here is an example of the use of this function (including using it to recreate `add1` and `add2` from above:

```
add1 = add_n(1)
add2 = add_n(2)

print(add2(3))  # prints 5
print(add1(7))  # prints 8
print(add_n(8)(9))  # prints 17
```

Try Now:

What type is each of the following values?

Back to Top

- `add_n`
- `add_n(7)`
- `add_n(9)(2)`
- `add_n(0.2)(3)`
- `add_n(0.8)(2)`

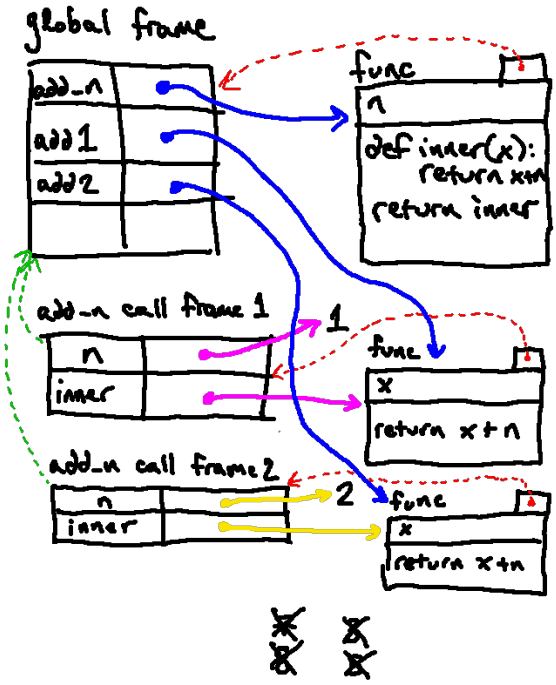
Show/Hide

- `add_n` is a `function`, as with other examples we saw before.
- `add_n(7)` is the result of calling `add_n` with `7` as its argument, which will also be a `function`.
- `add_n(9)(2)` calls `add_n` with an argument of `9`, and then it calls the *result* with an argument of `2`. This yields `11`, an `int`.
- `add_n(0.2)(3)` yields `3.2`, a `float`.
- `add_n(0.8)(2)` yields `2.8`, a `float`.

3.3) More Environment Diagrams

The examples above may be a little bit surprising, but we can understand them by working through them using an environment diagram (and even if they aren't surprising, it's important to know exactly what Python is doing under the hood.) Here, we'll look at simulating a piece of the above code using an environment diagram:

```
def add_n(n):
    def inner(x):
        return x + n
    return inner

add1 = add_n(1)
add2 = add_n(2)

print(add2(3))
print(add1(7))
```



Back to Top

<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 15

Here we see the results of cleaning up the most recent frame. Note that, because there are no more references to them, the `7` and `8` objects are garbage collected. At this point, we have reached the end of the program, so Python will exit.

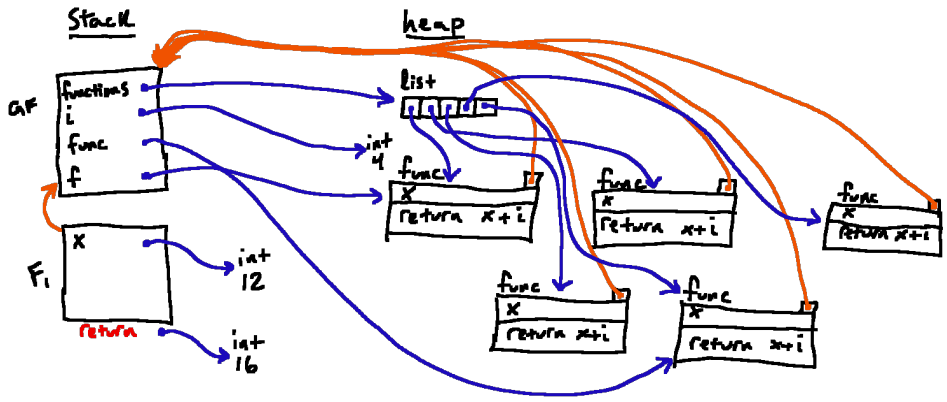
4) Understanding the Mystery Program

Earlier, we looked at the following piece of code as an example of code that is difficult to understand:

```
1 functions = []
2 for i in range(5):
3     def func(x):
4         return x + i
5     functions.append(func)
6
7 for f in functions:
8     print(f(12))
```

It is somewhat surprising that, despite the looping structure here, when we run this code, we see five `16`'s printed to the screen! Despite the surprising nature of this example, though, we now have all of the tools we need in order to make sense of this example and to understand *why* it behaves the way it does. We'll walk through an environment diagram to explain this behavior, and you're strongly encouraged to follow along (and to reach out for help if any of the steps are unclear!).

We'll start by drawing the diagram just for the first segment of the code (lines 1-5, where we are building up the `functions` list). Again, we encourage you to try to stay one step ahead of the drawings below (that is, try to draw out how things will change during each step, then click ahead and compare your work against our diagram).



Back to Top

[<< First Step] [< Previous Step] [Next Step >] [Last Step >>]

STEP 16

Looking up `x` inside of **F1**, we find the `12` that is bound locally. But `i` is *not* bound locally. So what do we do? We follow the parent pointer, and we *do* find the name `i` in the global frame. It references a value of `4` for `i`, so that's what we'll use.

Then we add those two values together to get a new `int` object representing `16`, which we return.

We'll stop here with the diagram, but note that this result would have been the same regardless of which of these function objects we called. None of them remembers the value that `i` held when it was created; they all simply say to look up the *current* value of `i` and add it to their inputs! So as we continue to loop and call each of these function objects in turn, they all produce the same output!

4.1) Closures

Now that we've explained the interesting phenomenon from the mystery program, we'd like to try to "fix" it (presumably, the person who wrote that code did not intend to see five `16`'s, but rather some number that is changing, i.e., the intent was probably to create five noticeably different functions: one that adds `0` to its input, one that adds `1` to its input, one that adds `2` to its input, and so on...). Before we can get there, though, we're going to introduce one more bit of terminology. This section is not about introducing a new *rule* for how function objects behave (we've already covered all of them, in fact!) but rather a powerful effect of those rules.

Importantly, a function object "remembers" the frame in which it was defined (its enclosing frame), so that later, when the function is being called, it has access to the variables defined in that frame and can reference them from within its own body. We call this combination of a function and its enclosing frame a **closure**, and it turns out to be a really useful structure, particularly when we define functions inside the bodies of other functions (like the `add_n` example from above.)

4.2) Fixing the Mystery Code

Using this idea of a closure, we can fix the mystery code from earlier! The code below resolves the issue (at least insofar as preventing the output from all of the functions from being identical!) by *evaluating* `i` each time through the loop and setting up a *closure* for each of the functions we add to the `functions` list, such that, when each is called, it has access to a variable storing the value that `i` had when that function was created.

```
1 def add_n(n):
2     def inner(x):
3         return x + n
```

```
4       return inner
5
6   functions = []
7   for i in range(5):
8       functions.append(add_n(i))
9
10  for f in functions:
11      print(f(12))
```

Back to Top

When this program is run, what will its output be?

Submit View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

5) A Note About Aliasing

Aliasing is good! Except when it's not. Be careful of when you want aliases and when you don't.

With multiple frames, aliasing can be trickier to notice (aliases might be in different environments!)

Try Now:

Consider the example below:

Back to Top

```
def double(nums):
    # Given a list of numbers, make a new list that doubles each number
    for i in range(len(nums)):
        nums[i] = nums[i] * 2
    return nums

global_nums = [1, 2, 3, 4]
print(double(global_nums))
print(global_nums)
```

What will this code output? Why?

Show/Hide

```
[2, 4, 6, 8]
[2, 4, 6, 8]
```

This may be unexpected that the global_nums changed, but with an environment diagram we can see that the nums variable inside the double function aliases the global_nums list. When we mutate the nums list by reassigning each index, we mutate global_nums as well.

To fix this, we need to create a new list as follows:

```
def double(nums):
    # Given a list of numbers, make a new list that doubles each number
    new_nums = []
    for i in range(len(nums)):
        new_nums.append(nums[i] * 2)
    return new_nums

global_nums = [1, 2, 3, 4]
print(double(global_nums))  # [2, 4, 6, 8]
print(global_nums)  # [1, 2, 3, 4]
```

Now because we create new_nums in the local frame, every time we call double a new list is created and modified, and we only use the input nums as a reference without modifying it.

6) Default and Keyword Arguments

For functions we've seen so far, we indicate the arguments by positions. For example, with this function:

```
def divide_twice(a, b):
    return (a/b)/b
```

Back to Top

When we call `divide_twice(12,2)`, python knows that `a` should be `12` and `b` should be `2` because that's the order we defined the arguments to come in.

However, there is another way to pass in arguments, using the name instead of the position. For example, we write the argument name, an equal sign, and the value we want it to take on.

```
divide_twice(b = 2, a = 12) # this would still be 3
```

Finally, there's a way to specify a function to have *optional* arguments. We signify these arguments with a variable name as usual, but we also add an equal sign and a default value. For example, if we wanted our function to have the option of printing the result before returning, we could add the optional argument `print_result`.

```
def divide_twice(a, b, print_result=False):
    answer = (a/b)/b
    if print_result:
        print(answer)
    return answer
```

Now we can still call `divide_twice` like we did before. If we don't specify a value for `print_result` it will be `False` by default as we indicated in the function definition.

```
divide_twice(12, 2) # prints nothing since print_result is False
```

But we can also specify a value for `print_result`. For example:

```
divide_twice(12, 2, True) # this will print 3 since print_result is now True.
```

It's common practice to use keyword specification for optional arguments because if there are multiple default arguments, it's not immediately clear which ones are being set. For example, we could explicitly show that `print_result` is a default argument being set to True as such:

```
divide_twice(12, 2, print_result=True)
```

7) Assert statements

So far, we have debugged and tested our programs mainly with print statements. However, Python comes with additional tools to help us test whether our code actually does what we intend.

Assert statements check a conditional statement. If the statement evaluates to True, the program continues as normal, but if it evaluates to False an AssertionError will be raised and stop the program.

```
>>> assert 5 > 4 # evaluates to True, does nothing
>>> assert 5 < 4
...
AssertionError
```

Back to Top

For example, we could test the `add_s` function below with assert statements with the following code:

```
def add_s(words):
    return [word + "s" for word in words]

if __name__ == '__main__':
    assert add_s(['can', 'add', 's']) == ['cans', 'adds', 'ss']
    assert add_s(['']) == ['s']
    assert add_s([]) == []
    print("done testing")
```

While this program only prints `done testing`, it also silently checks that the output matches what we expect, saving us from the hassle of manually checking whether the printed output is correct or not.

We can also use assert statements within functions to check that the input is valid:

```
def square(num):
    assert type(num) == float or type(num) == int, f"Expected float or int, got {num} which is of
{type(num)}."
    return num ** 2

print(square(5))
print(square("uh oh"))
```

Outputs:

```
25
...
AssertionError: Expected float or int, got uh oh which is of <class 'str'>.
```

Assert statements can slow down programs slightly but are generally good practice, especially for testing and debugging.

8) Generating Graphs with matplotlib

The `matplotlib.pyplot` module provides a number of useful functions for creating plots with Python. In this section we'll go over a few examples of how to generate different plots.

To import the `pyplot` module, add the following to the top of your script:

```
import matplotlib.pyplot as plt
```

Once you have done so, you can make a new plot by calling `plt.figure()` with no arguments. After that, you can use various functions to add data to the figures. When you are ready, calling the `plt.show()` function with no arguments will cause `matplotlib` to open windows displaying the resulting graphs. You can also add a legend and/or a title to the plot, aBack to Top
labels to the axes, as shown in the example below.

The following code will cause four windows to be displayed. Try running the code below on your own machine to see the results. Notice that the `plt.show()` function does not return until the plotting windows are closed.

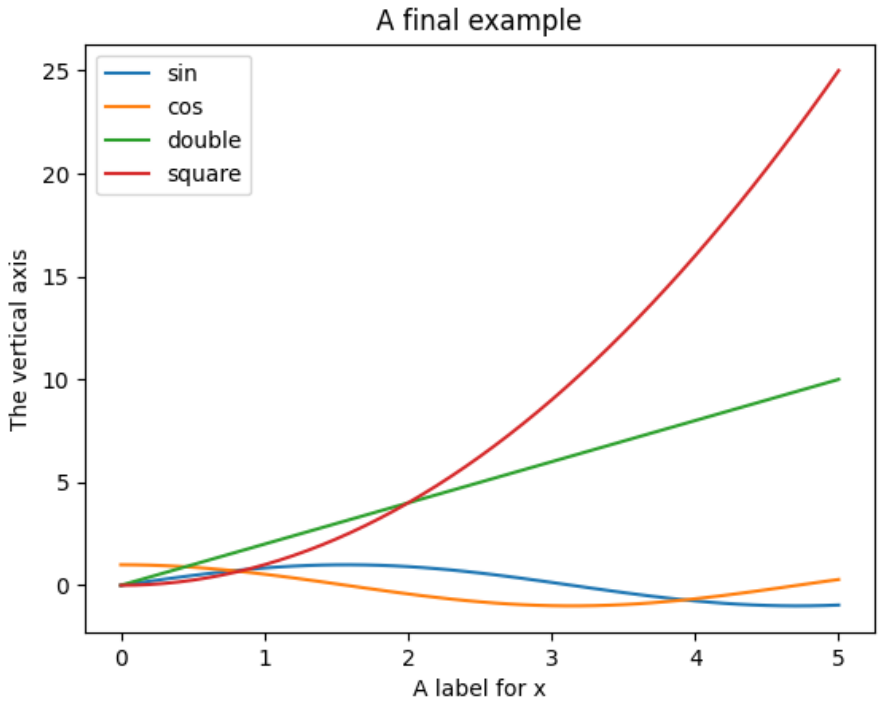
```
import matplotlib.pyplot as plt
import numpy as np

# here we plot a set of "y" values only; these are associated automatically
# with integer "x" values starting with 0.
plt.figure()
plt.plot([9, 4, 7, 6])

# if given two arguments, the first list/array will be used as the "x" values,
# and the second as the associated "y" values
plt.figure()
plt.plot([10, 9, 8, 7], [1, 2, 3, 4])
plt.grid()  # this adds a background grid to the plot

# we can also create scatter plots.  scatter plots require both "x" and "y"
# values.
plt.figure()
plt.scatter([10, 25, 37, 42], [12, 28, 5, 37], label='scatter')
# multiple calls to plt.plot or plt.scatter will operate on the same axes
plt.plot([10, 40], [5, 20], 'r', label='a line') # the 'r' means 'red'
plt.plot([5, 9, 15, 30], [10, 20, 30, 35], 'k', label='more data')
plt.legend()


plt.figure()

# generates 250 random points using a normal distribution
# with a mean of 170 and standard deviation of 10
x = np.random.normal(170, 10, 250)
plt.hist(x, bins=20, alpha = .5) # 20 bins, 50% transparency
plt.hist(np.random.normal(185, 10, 250), alpha = .5)
plt.title('A Histogram example')
plt.xlabel('A label for x')
plt.ylabel('The vertical axis')
plt.show()


# finally, display the results
print('Showing Graphs')
plt.show()
# Note that all figures need to be closed before the program prints Done
print('Done')
```

Using our graphing skills, we can now finally plot the graphs of the functions we defined earlier:

Back to Top

```
import matplotlib.pyplot as plt
import math

def response(f, lo, hi, step):
    # given a function f,
    # calculate and return a list of x
    # a list of and f(x) values
    x, y = [], []
    i = lo
    while i <= hi:
        x.append(i)
        y.append(f(i))
        i += step
    return x, y


sinx, siny = response(math.sin, 0, 5, 0.1)
cosx, cosy = response(math.cos, 0, 5, 0.1)
def double(x):
    return 2 * x
doublex, doubley = response(double, 0, 5, 0.1)
def square(x):
    return x**2
squarex, squarey = response(square, 0, 5, 0.1)

plt.figure()
plt.plot(sinx, siny, label='sin')
plt.plot(cosx, cosy, label='cos')
plt.plot(doublex, doubley, label='double')
plt.plot(squarex, squarey, label='square')
plt.title('A final example')
plt.xlabel('A label for x')
plt.ylabel('The vertical axis')
plt.legend()
plt.show()
print("done")

```

Running the code above will produce a graph like the one below:



Back to Top

9) Summary

In this set of readings, we revisited the details of how Python invokes functions. We also learned the ways in which Python functions are *first-class objects*. They can be treated just like any other objects in Python: among other things, they can be passed as arguments to functions and can be returned as the result of other functions! And we saw `assert`, which can be used to test programs automatically.

In next week's readings, we'll investigate one way to *use* functions: recursion. And we'll talk about strategies for designing large programs.