

Function Definitions

15-110 – Friday 09/10

Announcements

- Feedback is now released for Check1
 - To view your feedback, open your assignment in Gradescope, then click on the question name on the right sidebar that you want to see feedback for.
 - Note that *all* rubric items are displayed by default; the rubric items applied to *your* submission should be highlighted.
 - If you find a grading error, use the Request Regrade button to ask the Lead TAs to take a second look
- Hw1 is due **Monday at noon**
 - For Hw1 – Programming, work primarily in the **editor**, not the interpreter!

Learning Objectives

- Use **function definitions** when reading and writing algorithms to implement procedures that can be repeated on different inputs
- Recognize the difference between **local** and **global scope**
- Trace the **call stack** to understand how Python keeps track of nested function calls

Function Definitions

Function Definitions Run on Abstract Input

Now that we have all the individual components of functions, we can write new function definitions ourselves.

To write a function, you need to determine what **algorithm** you want to implement. You'll convert that algorithm into code that runs on **abstract input**.

Core Function Definition

Let's start with a simple function that has no explicit input or output; instead, it has a side effect (printed lines).

```
def helloWorld():  
    print("Hello World!")  
    print("How are you?")
```

```
helloWorld()
```

`def` is how Python knows the following code is a function definition

`helloWorld` is the **name** of the function. This is how we'll call it.

The **colon** at the end of the first line, and the **indentation** at the beginning of the second and third, tell Python that we're in the **body** of the function.

The body holds the algorithm. When the indentation stops, the function is done.

In this example, the last line **calls** the function we've written.

Parameters are Abstracted Arguments

To add input to the function definition, add **parameters** inside the parentheses next to the name.

These parameters are variables that are not given initial values. Their initial values will be provided by the arguments given each time the function is called.

```
def hello(name):  
    print("Hello, " + name + "!")  
    print("How are you?")
```

```
hello("Stella")  
hello("Dippy")
```

Return Provides the Returned Value Output

To make our function have a non-`None` output, we need to have a **return statement**. This statement specifies the value that should be substituted for the function call when the function is called on a specific input.

```
def makeHello(name):  
    return "Hello, " + name + "! How are you?"  
  
s = makeHello("Scotty")
```

As soon as Python returns a value, it exits the function. Python ignores any lines of code after a return statement.

Activity: Write a Function

You do: write a function `convertToQuarters` that takes a number of dollars and converts it into quarters, returning the number of quarters.

For example, if you call `convertToQuarters` on `2` (\$2), the function should return `8` (8 quarters).

Control Flow

Writing code with function definitions introduces a new concept to our programs – **control flow**. This is the order that statements are executed in as we run a program.

Before, all our programs ran sequentially from the first statement to the last. But with function definitions, Python will need to **redirect** the control flow whenever we call a function that we've defined.

Control flow is an incredibly useful tool, but it also makes it more difficult to read and comprehend a program. In particular, when you read code with a function definition, you have to keep in mind that that definition will not influence the program until it is called.

Analyzing Functions

When a function you've defined **is** called, you can figure out what it will evaluate to by tracing through the definition.

```
def addTip(cost, percentToTip):  
    return cost + cost * percentToTip  
  
total = addTip(20.00, 0.17)
```

For example, in this function call, we know the inputs (**20.00** and **0.17**), so the output must be **20.00 + 20.00 * 0.17**, which is **23.4**.

Note that this only works because we defined **addTip** **before** we called it! Python will still execute all the statements in order.

Activity: Analyze the function

You do: what are the arguments and returned value of this function call, given the definition?

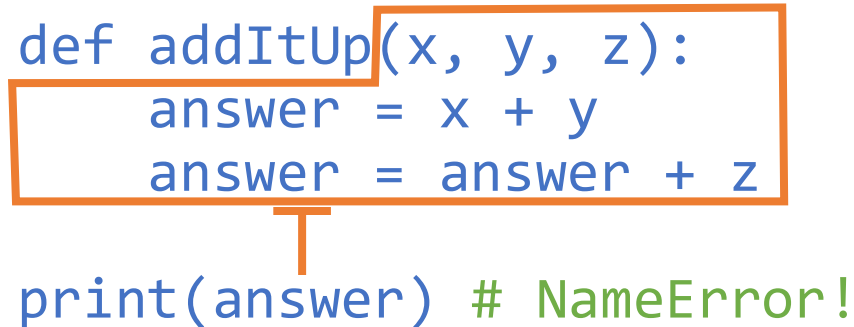
```
def distance(x1, y1, x2, y2):  
    xPart = (x2 - x1)**2  
    yPart = (y2 - y1)**2  
    print("Partial Work:", xPart, yPart)  
    return (xPart + yPart) ** 0.5  
  
result = distance(0, 0, 3, 4)
```

Scope

Variables Have Different Scopes

All the work done in a function is only accessible in that function. In other words, if we make a variable in a function, the outer program can't access it; the only way to transmit its value is to return it.

```
def addItUp(x, y, z):  
    answer = x + y  
    answer = answer + z  
  
print(answer) # NameError!
```

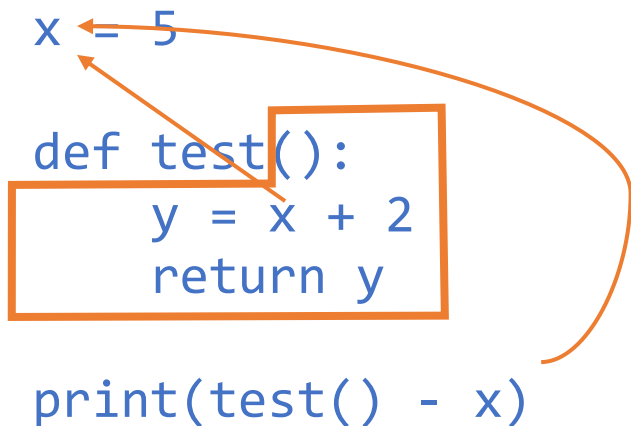
A diagram consisting of an orange rectangular box that encloses the function definition and its two lines of code. A vertical orange line extends downwards from the bottom center of this box to the 'answer' variable in the print statement below. This visualizes that the 'answer' variable is only defined within the function's local scope and is not accessible in the global scope where the print statement is executed.

The variable `answer` has a **local scope** and is accessible only within the function `addItUp`.

Everything Can Access Global Variables

On the other hand, if a function is told to use a variable it hasn't defined, the function automatically looks in the **global scope** (outside the function at the **top level**) to see if the variable exists there.

```
x ← 5  
  
def test():  
    y = x + 2  
    return y  
  
print(test() - x)
```



If you change a global variable in a function, that's a **side effect**! It's unlikely that you'll want to use this, but good to know for debugging.

Scope is Like Names

You can think of the scope of a variable as being like its last name. For example, consider the following code:

```
x = "bar"
```

```
def test():  
    x = "foo"  
    print("A", x)
```

```
test()  
print("B", x)
```

`x` exists in both the local and the global scope, but the two `x` variables are **separate** and have different values.

Analogy: knowing two people both named Andrew. They have the same first name, but **different last names**.

In the code above, the last name of the function's `x` would be *test*, while the last name of the top-level `x` would be *global*.

In general, it's best to keep variable names different to avoid confusion.

Activity: Local or Global?

Which variables in the following code snippet are global? Which are local?
For the local variables, which function can see them?

```
name = "Farnam"

def greet(day):
    punctuation = "!"
    print("Hello, " + name + punctuation)
    print("Today is " + day + punctuation)

def leave():
    punctuation = "."
    print("Goodbye, " + name + punctuation)

greet("Monday")
leave()
```

Function Call Stack

Function Calls in Function Definitions

It isn't too hard to trace a function call when it goes through a single definition, but it gets a lot harder when that definition **calls another function**.

When the code to the right calls the function `outer`, `outer` will run a bit of code, then call the function `inner`.

Python needs to keep track of which variables are in scope at any given point, and where returned values should be sent. It does this with a **call stack**.

We'll primarily use the call stack to help us trace and read code.

```
def outer(x):  
    y = x / 2  
    return inner(y) + 3
```

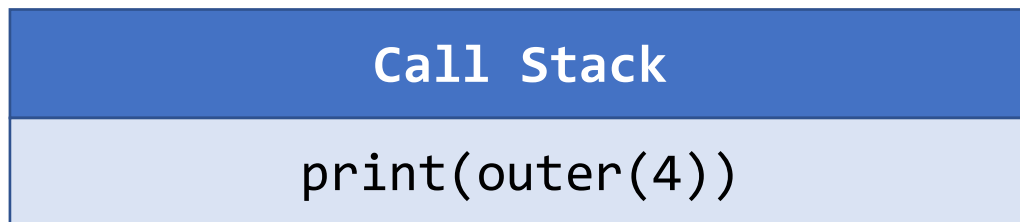
```
def inner(x):  
    y = x + 1  
    return y
```

```
print(outer(4))
```

Tracing the Code

When Python runs through this code, it adds `outer` to its state, then it adds `inner`.

When it reaches the last line, it must call `outer` to evaluate the expression. Python puts this line on the **stack** to keep track of where it was before.



```
def outer(x):  
    y = x / 2  
    return inner(y) + 3
```

```
def inner(x):  
    y = x + 1  
    return y
```

```
print(outer(4)) ←
```

Tracing the Code

Python traces through the outer function normally until it reaches the call to inner.

Now it needs to add **another** layer to the stack, to keep track of where it is in outer.

Call Stack
<code>print(outer(4))</code>
<code>return inner(2.0) + 3</code>

```
def outer(x):  
    y = x / 2  
    return inner(y) + 3 ←
```

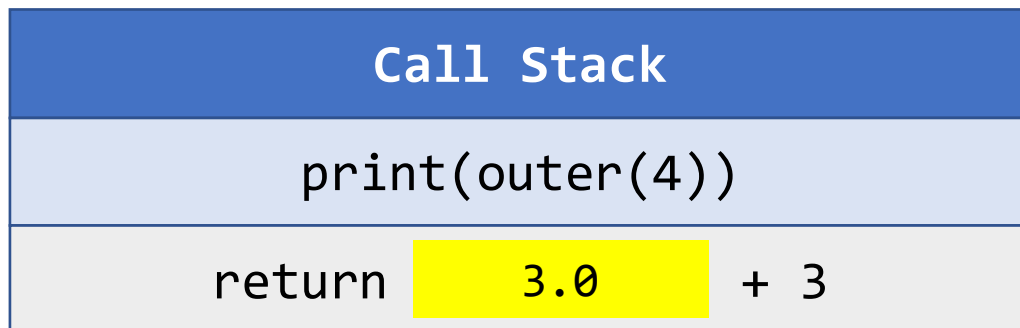
```
def inner(x):  
    y = x + 1  
    return y
```

```
print(outer(4)) ←
```

Tracing the Code

Python is able to fully execute inner without calling another function.

When it reaches the return statement, it looks to the most recent part of the stack to see where to go next. The returned value is substituted for the call there.



```
def outer(x):  
    y = x / 2  
    return inner(y) + 3 ←
```

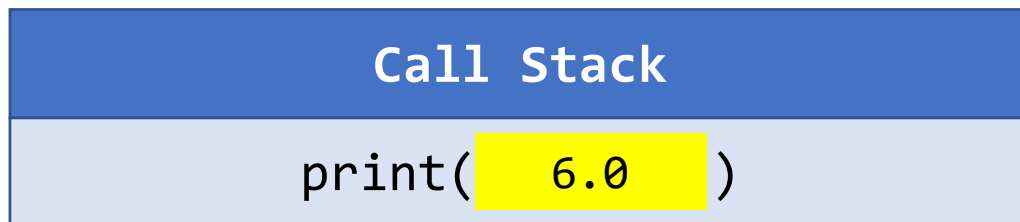
```
def inner(x):  
    y = x + 1  
    return y
```

```
print(outer(4)) ←
```

Tracing the Code

When the value has been returned, that layer is **removed** from the stack.

Python is able to finish running the outer function, and the return statement goes back to the first layer of the stack. We'll then print `6.0` and be done!



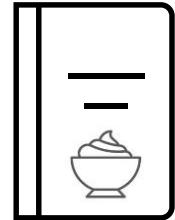
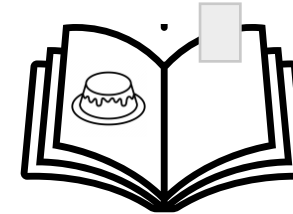
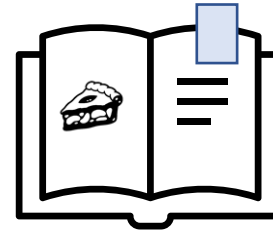
```
def outer(x):  
    y = x / 2  
    return inner(y) + 3
```

```
def inner(x):  
    y = x + 1  
    return y
```

```
print(outer(4)) ←
```

Analogy: Baking with Bookmarks

You can think of the call stack like a series of **bookmarks** that help you keep your place as you trace the code.



For example, perhaps I'm following a recipe to make an apple tart. One step of the recipe tells me to make a frangipane (custard), but I don't know how to do that!

I can put a bookmark on my current step and find another cookbook with a recipe for making frangipane, then start following that recipe.

Maybe that recipe tells me to cream the butter and sugar, and I have to look in yet another cookbook to learn how to do that. Each new recipe is another layer on the stack.

Call Stack
<code>makeAppleTart(ingredients)</code>
<code>makeFrangipane(subIngredients)</code>
<code>creamButterSugar(butter, sugar)</code>

[if time] Activity: Trace the Function Calls

You do: given the code to the right, use a call stack to trace through the execution of the code.

It can be helpful to jot down the current variable values as well, so you don't have to hold them all in your head.

What will be printed at the end?

```
def calculateTip(cost):  
    tipRate = 0.2  
    return cost * tipRate
```

```
def payForMeal(cash, cost):  
    cost = cost + calculateTip(cost)  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 8.00)  
print("Money remaining:", wallet)
```

Call Stacks in Error Messages

Call stacks will show up naturally in your code whenever you encounter an **error message**.

The call stack shows you exactly which function calls led to the location where the error occurred.

If we insert an error into the middle of the code, you can see how each level of the stack is listed out.

```
def outer(x):  
    y = x / 2  
    return inner(y) + 3  
  
def inner(a):  
    b = a + 1  
    print(oops) # will cause an error  
    return b  
  
print(outer(4))
```

```
Traceback (most recent call last):  
  File "example.py", line 10, in <module>  
    print(outer(4))  
  File "example.py", line 3, in outer  
    return inner(y) + 3  
  File "example.py", line 7, in inner  
    print(oops) # will cause an error  
NameError: name 'oops' is not defined
```

Learning Objectives

- Use **function definitions** when reading and writing algorithms to implement procedures that can be repeated on different inputs
- Recognize the difference between **local** and **global scope**
- Trace the **call stack** to understand how Python keeps track of nested function calls

Feedback: <https://bit.ly/110-f21-feedback>