Function Definitions

15-110 – Friday 09/06

Announcements

- Hw1 is due Monday at noon
- Email both of us if you're still on the waitlist!
- Next week: First Quizlet
  - There are 8 (possibly 9) quizlets throughout the semester on Wednesdays
    - Lowest two scores dropped
  - Procedure:
    - Bring a piece of paper
    - You'll have 5 minutes to answer the question displayed on the screen
    - No computers, phones, notes, or collaboration
    - When time is up, take a picture and upload to Gradescope
    - Demo next time

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 function calls to understand how Python keeps track of **nested function calls**

Function Definitions

A **function** is a code construct that represents an algorithm.

```
pet1 = "Spot"
pet2 = "Stella"
pet3 = "Willow"

print("See " + pet1 + ". See " + pet1 +
      " run. Run, " + pet1 + ", run!")

print("See " + pet2 + ". See " + pet2 +
      " run. Run, " + pet2 + ", run!")

print("See " + pet3 + ". See " + pet1 +
      " run. Run, " + pet3 + ", run!")
```

=

```
def outputPetName(pet):
    print("See " + pet + ". See " + pet +
      " run. Run, " + pet + ", run!")

outputPetName("Spot")
outputPetName("Stella")
outputPetName("Willow")
```

We **define** a function once, then **call** it many times.

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

```
1  def helloWorld():
2      print("Hello World!")
3      print("How are you?")
4
5  helloWorld()
6
```

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

```
1 def helloWorld():
```

`def` is how Python knows this is a function definition

`helloWorld` is the **name** of the function

```
1 def helloWorld():
2     print("Hello World!")
```

indentation (tab)

`:` and indentation is start of **function body**

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

indentation

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

indented lines are **function body** which holds the algorithm

when the indentation stops, the function is done

calling the function we defined

```
1 def helloWorld():
2     print("Hello World!")
3     print("How are you?")
4
5 helloWorld()
6
```

We can define a function with **parameters** by putting the variable names of the parameters **inside the parentheses**.

```
1  def hello(name):
2      print("Hello, " + name + "!")
3      print("How are you?")
4
5  hello("Kimchee")
6  hello("Stella")
```

`name` is a variable inside the function that we can use to do operations **inside the function body**.

We specify a function's **returned value** by writing a **`return` statement**.

```
1  def makeHello(name):
2      return "Hello, " + name + "! How are you?"
3
4  s = makeHello("Scotty")
5
```

Is there is no return statement, the **function returns None**!

```
1  def addNumbers(x,y):
2      z = x+y
3
4  returnedValue = addNumbers(1,2)
5
```

`returnedValue` is None!

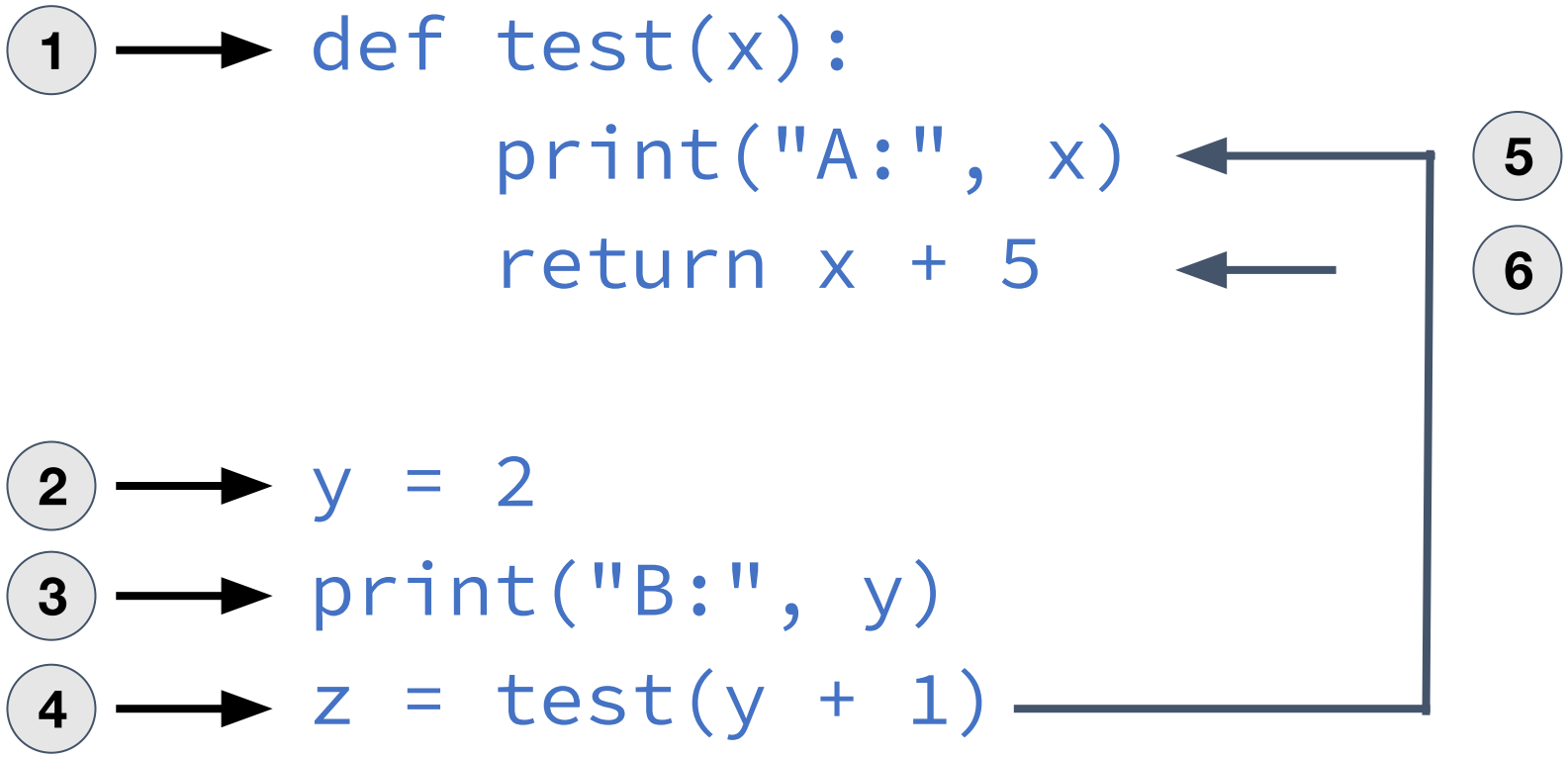
Activity: Write a Function

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).

```
x = convertToQuarters(2)
```

Control flow is the order that **statements are executed** as we run a program.



When you read code with a function definition, that definition **will not influence the program until it is called**!

Example Code

For example, what will be printed when we run the following code?

```
def test(x):
    print("A:", x)
    return x + 5

y = 2
print("B:", y)
z = test(y + 1)
```

Interpreter:
B: 2
A: 3

We do not enter the function until it is called. That means B is printed before A, even though its line occurs further down in the code!

Activity: Analyzing functions

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

```
def addTip(cost, percent):
    tip = cost * percent
    print("Tip:", tip)
    return cost + tip

total = addTip(25, 0.2)
```

Scope

x is a **local variable**:
it is only accessible **within the function** `test`

```
def test(x):
    print("A:", x)
    return x + 5


y = 2
print("B:", y)
z = test(y + 1)
```

x is a **global variable**:
it is **accessible everywhere** after it is defined (even inside functions!)

```
x = 5

def addTwo():
    y = x + 2
    return y

print(addTwo() - x)
```

Python lets us do **weird confusing stuff** like this:

```
x = 5

def test():
    x = 2
    print("A", x)

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

What does this print?

We can **visualize code execution** with pythontutor.com!

```
x = 5                          global x

def test():
    x = 2                      local x
    print("A", x)

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

This is an excellent learning tool that is completely free to use.

Activity: Local or Global?

Which variables are **global**? Which are **local**?

For the local variables, **which function can see** them?

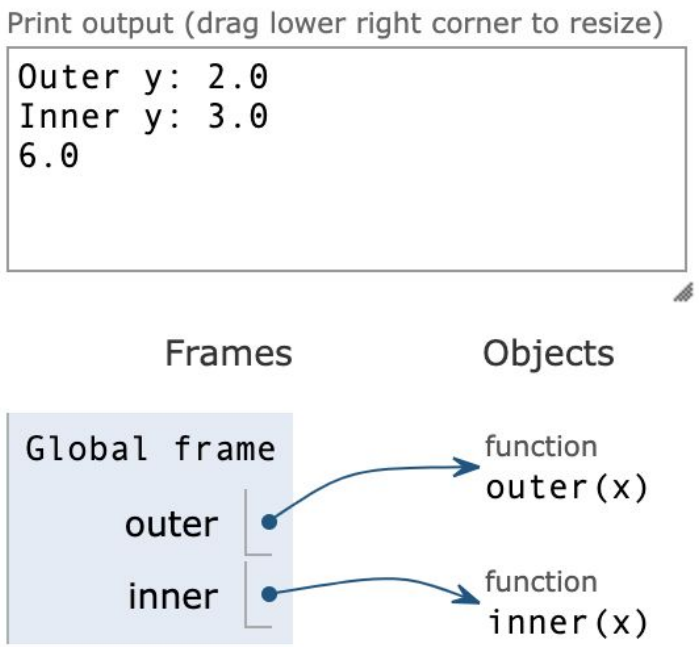
```
name = "Farnam"

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

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

greet("Friday")
leave()
```

Function Call Tracing

It gets a lot harder to trace functions when **a function definition calls another function**.

```
def outer(x):
    y = x / 2
    print(“Outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("Inner y:", y)
    return y

print(outer(4))
```

Check this out in pythontutor.com.



Tracing the Code

Interpreter:

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

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

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

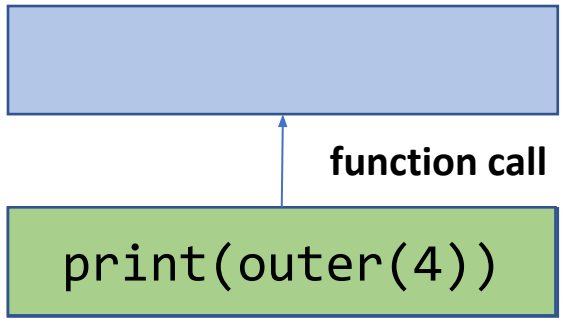
```
outer function
inner function
```

Tracing the Code

```
Interpreter:
```

When it reaches the last line, it must call outer to evaluate the expression.

The computer puts a 'bookmark' on the line it was on so it won't lose its place.




```
outer function
inner function
```

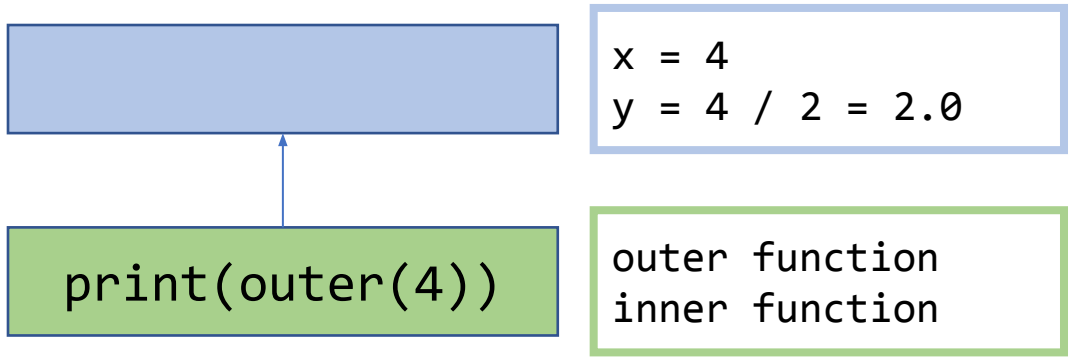
```
def outer(x):
    y = x / 2
    print("outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

Tracing the Code

Interpreter:
```
outer y: 2.0
```

Python traces through the outer function normally, keeping track of the local state, until it reaches the call to inner.

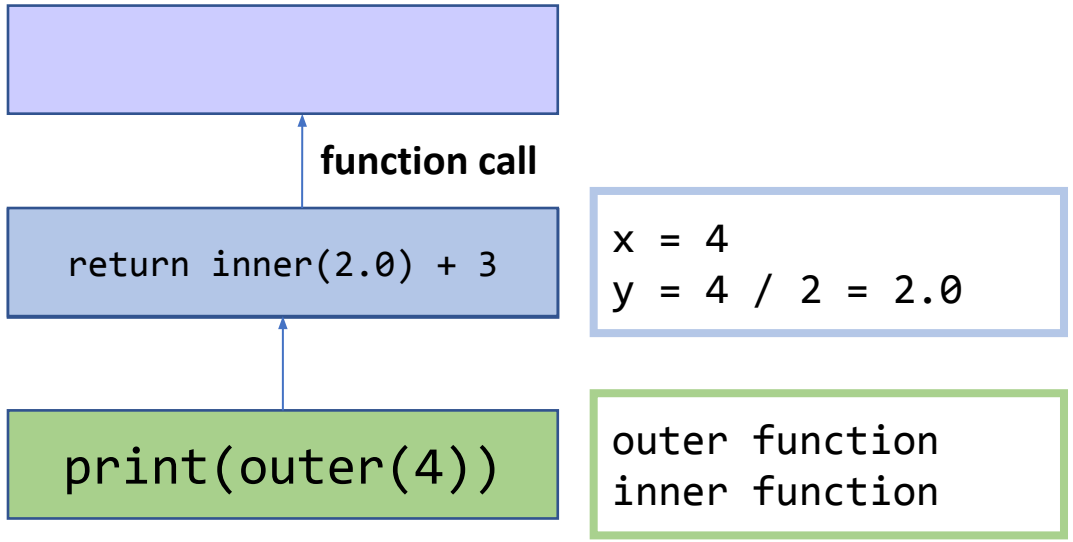
```
def outer(x):
    y = x / 2
    print("outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```



Tracing the Code

```
Interpreter:
outer y: 2.0
```

Once again, Python leaves a 'bookmark' at its current location, then moves to the inner function to set up a new local state.



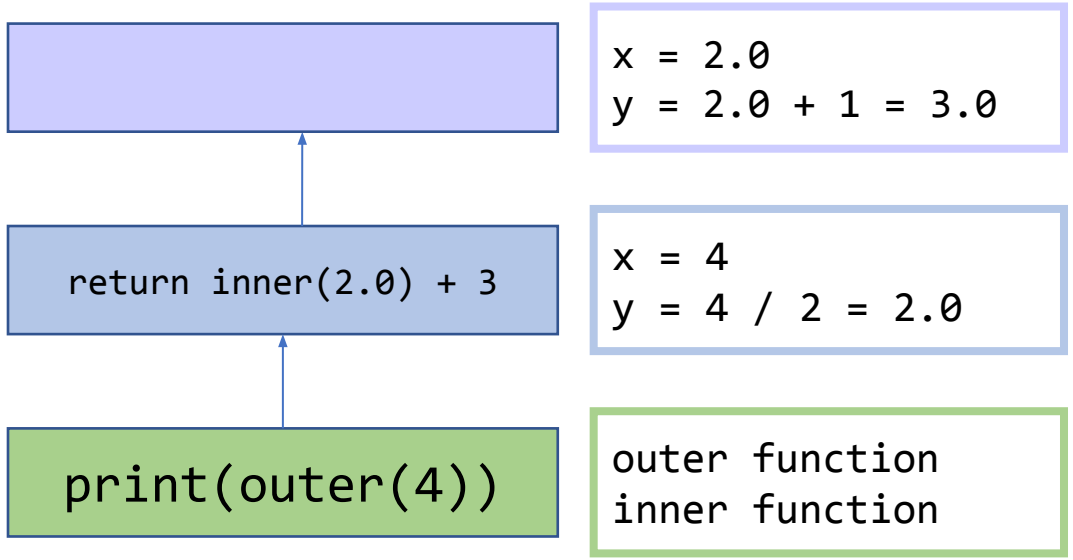

```
def outer(x):
    y = x / 2
    print("outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

Tracing the Code

```
Interpreter:
outer y: 2.0
inner y: 3.0
```

Python can fully execute inner without calling another function.

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

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

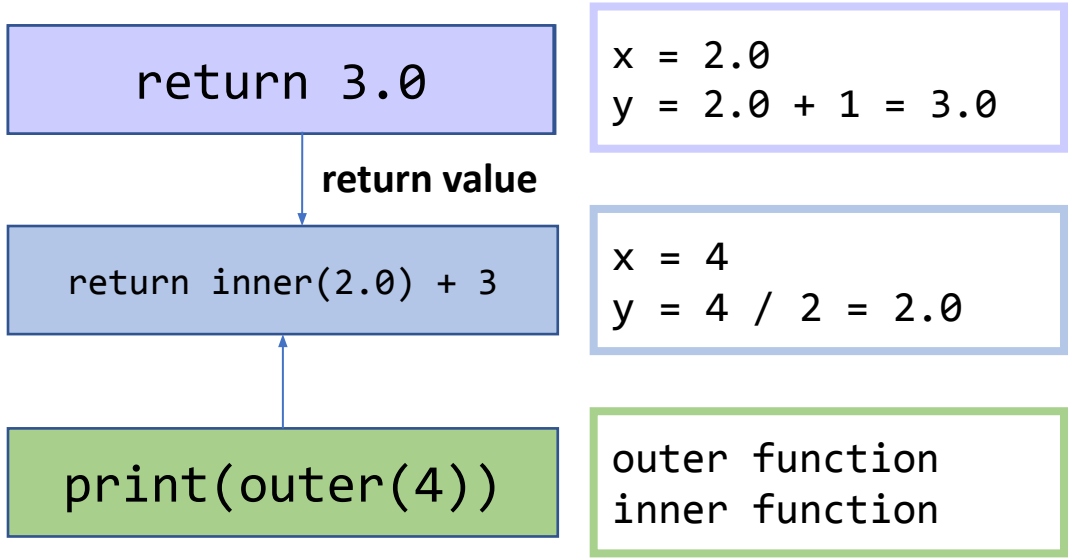


Tracing the Code

```
Interpreter:
outer y: 2.0
inner y: 3.0
```

When Python reaches the return statement of inner, it returns 3.0 to the function that previously called it, outer, by checking the bookmark.

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

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```



Tracing the Code

```
Interpreter:
outer y: 2.0
inner y: 3.0
```

When the value 3.0 is returned, it takes the place of the function call expression.

Now Python can finish running the outer function.

```
return 3.0 + 3
```

```
x = 4
y = 4 / 2 = 2.0
```

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

```
outer function
inner function
```

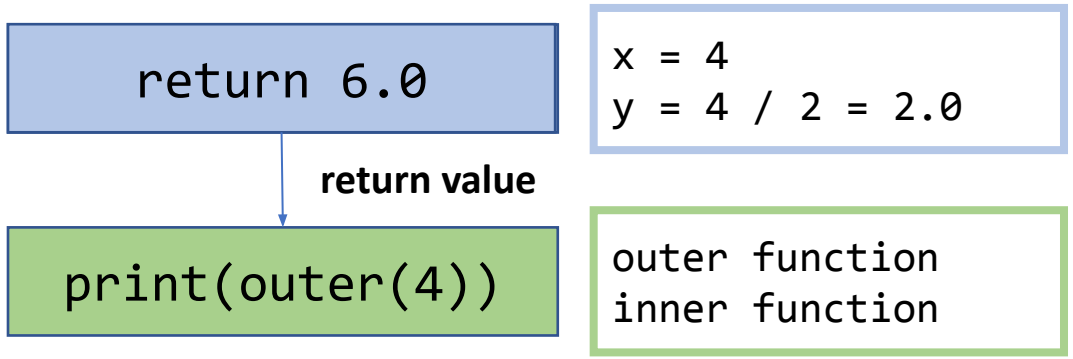
```
def outer(x):
    y = x / 2
    print("outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

Tracing the Code

```
Interpreter:
outer y: 2.0
inner y: 3.0
```

When outer finishes, it returns 6.0 to the next bookmarked function, the original call.

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

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```



Tracing the Code

```
Interpreter:
outer y: 2.0
inner y: 3.0
6.0
```

6.0 takes the place of outer(4), the value is printed, and the code is done!

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

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

```
print(6.0)
```

```
outer function
inner function
```

Function Calls in Error Messages

Function call 'bookmarks' will show up naturally in your code whenever you encounter an error message.

The lines of the error message show 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 'bookmark' 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 "C:\Users\river\Downloads\example.py", line 10, in <module>
    print(outer(4))
  File "C:\Users\river\Downloads\example.py", line 3, in outer
    return inner(y) + 3
  File "C:\Users\river\Downloads\example.py", line 7, in inner
    print(oops) # will cause an error
NameError: name 'oops' is not defined
```

[if time] Activity: Trace the Function Calls

You do: given the code to the right, trace through the execution of the code and the function calls.

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)
```

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 function calls to understand how Python keeps track of **nested function calls**