Function Definitions

15-110 – Friday 1/26

Announcements

- Hw1 is due Monday at noon

- Next week: First Quizlet
  - There are 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

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).

```python
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.

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. Use a function's name to call it.

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.

```python
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.

```python
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.

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!

Scope

Variables Have Different Scopes

```
def averageOfThree(x, y, z):
    total = x + y + z
    return total / 3

print(averageOfThree(1, 4, 10)) # 5.0
print(total) # NameError!
```
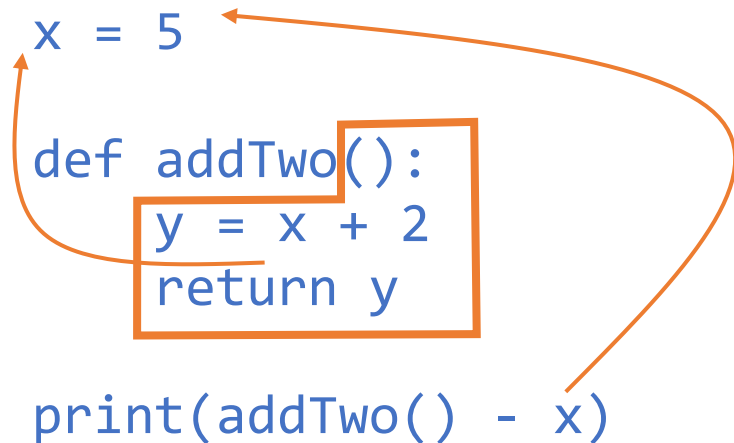
The variable total has a **local scope** and is accessible only within the function averageOfThree. Note that the parameters x, y, and z also have local scope, as they must be assigned values in a function call before we can use them.

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 addTwo():
    y = x + 2
    return y

print(addTwo() - x)
```

It's like a one-way mirror. Functions can see global variables, but global-level code cannot see local variables.

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 = 5

def test():
    x = 2
    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?

```python
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

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)
```

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.

```python
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

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

```python
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

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.

```
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))
```

function call

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

```
outer function
inner function
```

24

Tracing the Code

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

```python
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))
```

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

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

```
outer function
inner function
```

Tracing the Code

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

```python
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))
```

function call

```
return inner(2.0) + 3
```

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

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

```
outer function
inner function
```

26

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))    ←
```

```
x = 2.0
y = 2.0 + 1 = 3.0
```

```
return inner(2.0) + 3
```
```
x = 4
y = 4 / 2 = 2.0
```

```
print(outer(4))
```
```
outer function
inner function
```

27

Tracing the Code

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))     ⬅
```

```
return 3.0
```

```
x = 2.0
y = 2.0 + 1 = 3.0
```

return value

```
return inner(2.0) + 3
```

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

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

```
outer function
inner function
```

Tracing the Code

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

Now Python can finish running the outer 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))   ←
```

```
return 3.0 + 3
```

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

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

```
outer function
inner function
```

29

Tracing the Code

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
```

return 6.0

x = 4 y = 4 / 2 = 2.0

return value

print(outer(4))

outer function inner function

```
print(outer(4))  ⟵
```

Tracing the Code

`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

Analogy: Baking with Bookmarks

Function call tracing is 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 function call.
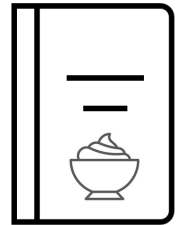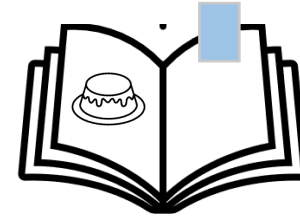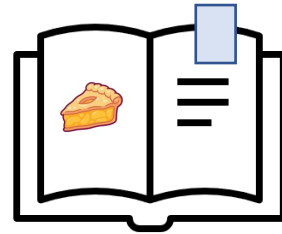
```
creamButterSugar(butter, sugar)
```

calls

```
makeFrangipane(subIngredients)
```

calls

```
makeAppleTart(ingredients)
```

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.

```python
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?

```python
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**