

CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida

epmikida@buffalo.edu

208 Capen Hall

Day 04

Function Definitions and Modules

Announcements

Schedule is now up on the course website

Office hours start this week

Labs start next week

Recap

- Two new kinds of expressions: variables and function calls
- Variables are a name that has been assigned a value
- Functions are a group of statements
 - They take arguments as input, do some work, and return a value
- Statements don't have a value, they have an effect
 - Assignment statement is used to create a variable, by assigning it a value
- Python has a number of useful built-in functions
 - But what if we need more...

Modules

- In addition to built-in functions, python also has a number of libraries
 - These libraries define their own functions
 - If you import these libraries, you can use their functions
 - Modules may also define variables

Math module in python

```
import math
```

```
x = 12
```

```
y = math.sin(x)
```

```
z = math.cos(x)
```

```
r = 42
```

```
area = math.pi * r * r
```

Math module in python

The import statement
tells python to load a
particular library

```
import math
```

```
x = 12
```

```
y = math.sin(x)
```

```
z = math.cos(x)
```

```
r = 42
```

```
area = math.pi * r * r
```

Math module in python

The import statement tells python to load a particular library

```
import math
```

```
x = 12  
y = math.sin(x)  
z = math.cos(x)
```

Once you've imported a library, you can call its functions...

```
r = 42  
area = math.pi * r * r
```

...and use it's variables

Function Definitions

Function definitions have a **header** and a **body**

The general form of a function definition looks like:

```
def <name>( <parameter list>):  
    <statement 1>  
    <statement 2>  
    ...
```


Function Definitions

Function definitions have a **header** and a **body**

The general form of a function definition looks like:

```
def <name>( <parameter list> ):
    <statement 1>
    <statement 2>
    ...
```

This is the header



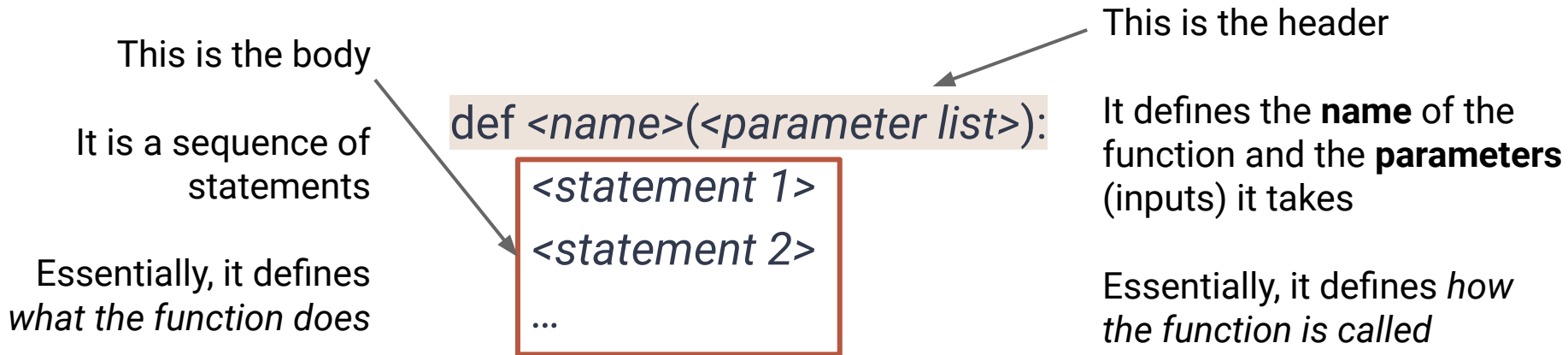
It defines the **name** of the function and the **parameters** (inputs) it takes

Essentially, it defines *how the function is called*

Function Definitions

Function definitions have a **header** and a **body**

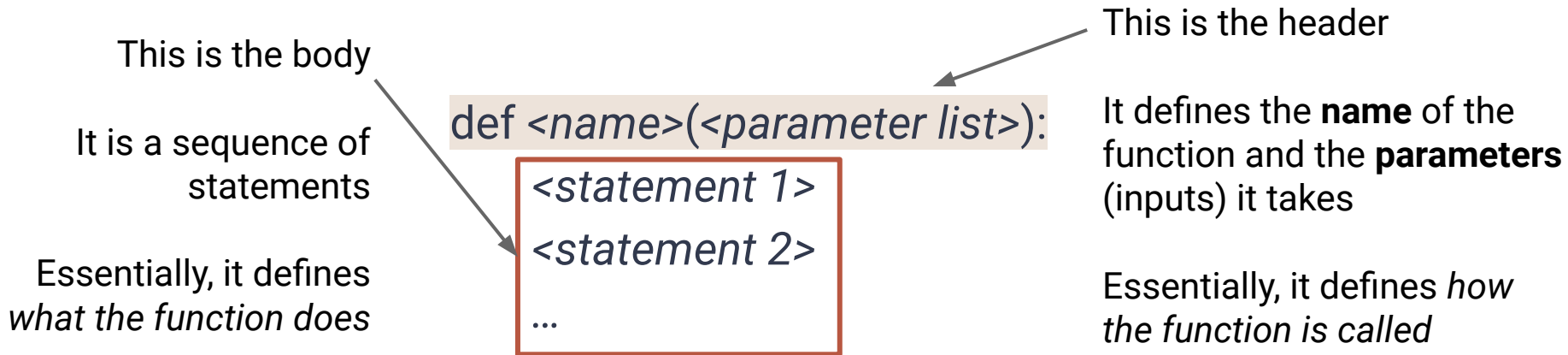
The general form of a function definition looks like:



Function Definitions

Function definitions have a **header** and a **body**

The general form of a function definition looks like:



Notice: The body is indented one level to the right of the header. This is required!

Now an Example

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

Now an Example

def is a **keyword**



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

Keywords are words reserved by python for specific uses. You cannot use them as a name.

A full list is [here](#)

Now an Example

averageOfThree is the name we have given this function. Function names follow the same rules as variable names.

def is a **keyword**



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

Keywords are words reserved by python for specific uses. You cannot use them as a name.

A full list is [here](#)

Now an Example

averageOfThree is the name we have given this function. Function names follow the same rules as variable names.

def is a **keyword** →

Keywords are words reserved by python for specific uses. You cannot use them as a name.

A full list is [here](#)

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

x, y, and z are the three **parameters** for this function.

They are **variables** that can be used in the function body, and get their values from the **arguments** given when the function is called.

Now an Example

averageOfThree is the name we have given this function. Function names follow the same rules as variable names.

def is a **keyword**

Keywords are words reserved by python for specific uses. You cannot use them as a name.

A full list is [here](#)

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

The commas, parenthesis, and colon are **delimiters**.

x, y, and z are the three **parameters** for this function.

They are **variables** that can be used in the function body, and get their values from the **arguments** given when the function is called.


Now an Example

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

Now an Example

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

The parameters
can be used in the
function body.



Now an Example

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

This is a **return statement**.

It is a statement that occurs to signify the end of a function, and the value that the function *returns*.

The parameters can be used in the function body.

Now an Example

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

This is a **return statement**.

It is a statement that occurs to signify the end of a function, and the value that the function *returns*.

return is a **keyword**

It is followed by any expression.
The value of the expression becomes the value of the function call!

The parameters can be used in the function body.

Demo in Replit

In depth execution example

What actually happens when we execute:

```
answer = averageOfThree(5+7, 8, 2)
```

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

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

In depth execution example


```
answer = averageOfThree(5+7, 8, 2)
```

To *execute* this statement, we first *evaluate* the expression on the right-hand side to get a value.

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


In depth execution example

answer = averageOfThree(5+7, 8, 2)



12 8 2

The expression on the right-hand side is a function call.

To evaluate the function call, we must first get values for each argument.

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

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

12 8 2

The arguments (*values*) are then stored in the functions parameters (*variables*).

These variables are stored in a table, or *environment*.

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

Name	Value
x	12
y	8
z	2

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

We then start executing the function body, using values from the current environment when evaluating variables, and adding to the environment as needed.

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

Name	Value
x	12
y	8
z	2

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

We then start executing the function body, using values from the current environment when evaluating variables, and adding to the environment as needed.


```
def averageOfThree(x, y, z):
```

```
    average = (x + y + z) / 3
```

```
    return average
```

Name	Value
x	12
y	8
z	2

Using values of x, y, and z, this expression evaluates to 7.3333



In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

We then start executing the function body, using values from the current environment when evaluating variables, and adding to the environment as needed.

```
def averageOfThree(x, y, z):
```

```
    average = (x + y + z) / 3
```

```
    return average
```

Using values of x, y, and z, this expression evaluates to 7.3333

Name	Value
x	12
y	8
z	2
average	7.3333

The value is assigned to the variable average, which we add to the current environment.

In depth execution example

answer = averageOfThree(5+7, 8, 2)

When we hit a *return statement* we evaluate the expression (based on the current environment).

That value becomes the value of our function call.

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

Name	Value
x	12
y	8
z	2
average	7.3333

In depth execution example

answer = averageOfThree(5+7, 8, 2)

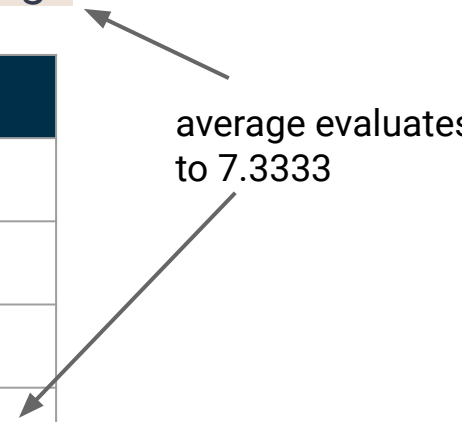
When we hit a *return statement* we evaluate the expression (based on the current environment).

That value becomes the value of our function call.

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

Name	Value
x	12
y	8
z	2
average	7.3333

average evaluates to 7.3333



In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

Therefore, our function call evaluates to 7.3333

When we hit a *return statement* we evaluate the expression (based on the current environment).

That value becomes the value of our function call.

```
def averageOfThree(x, y, z):
```

```
    average = (x + y + z) / 3
```

```
    return average
```

Name	Value
x	12
y	8
z	2
average	7.3333

average evaluates to 7.3333

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

```
def averageOfThree(x, y, z):
```

```
    average = (x + y + z) / 3
```

```
    return average
```

We can then finish executing our original assignment statement, which stores the value 7.3333 in the variable named answer.

Name	Value
answer	7.3333

In depth execution example

```
answer = averageOfThree(5+7, 8, 2)
```

```
def averageOfThree(x, y, z):
```

```
    average = (x + y + z) / 3
```

```
    return average
```

We can then finish executing our original assignment statement, which stores the value 7.3333 in the variable named answer.

Name	Value
answer	7.3333

Notice: Once the function execution ends, anything in the environment that was part of the function is removed!