ENGINEERING
TEXAS A&M UNIVERSITY

Week 11

Creating Functions

Learning Objectives

ENGINEERING
TEXAS A&M UNIVERSITY

Writing Functions, Scope

- Define a function without parameters, with parameters, and/or including default parameters
- Define a function with or without return variable
- Define a function utilizing another function
- Recognize when errors may or may not appear when defining or calling functions
- Define and explain the scope of a variable and where it will be recognized in a program
- Define local and global variables
- Account for how the mutable data within a list behaves with functions calls

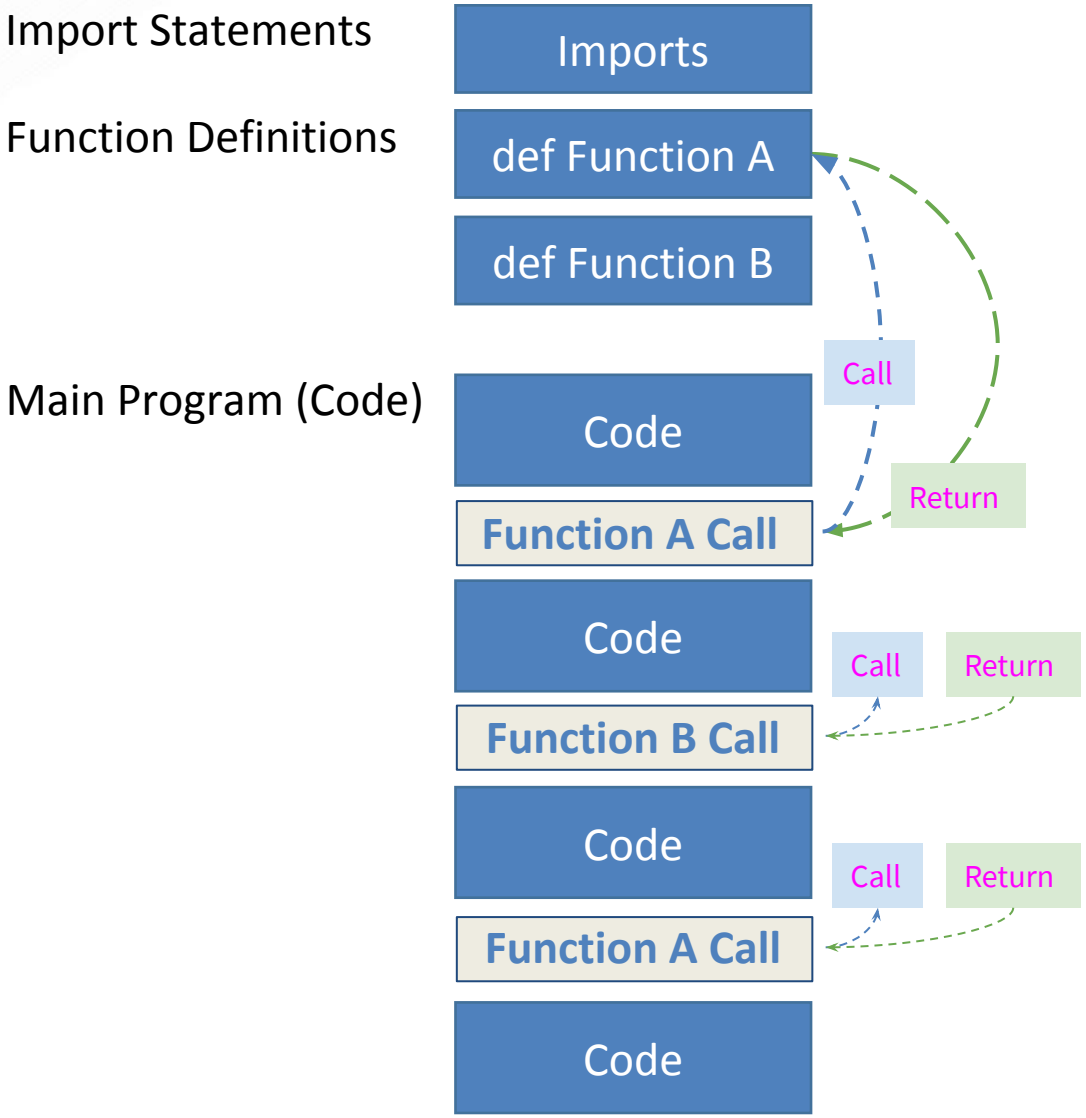
Function definition

```
def <function name>(<parameters>):
    <stuff to do>
```

Functions in a program

List <u>all</u> functions near the beginning of a program, before the “main” code.

1. import statements
2. function definitions
3. main code



Why do we have functions?

Passing Data To/From Functions

Functions are separate from the main code, and don't necessarily have access to the same data

Pass data into the function using **arguments**

- For example, `print("Howdy!")` passes "Howdy" into the function
- Ideally, all data the function needs should be passed in through the arguments

Return data from the function by a **return value**

- For example, `input()` returns a string entered by the user
- We can use this returned value in an expression, or assign it to a variable

What would happen here?

```
def warn():
    print("********** WARNING!!! **********")
    print("You are about to do something dangerous!")

def doublewarn():
    callingundefinedfunction()

warn()
```

Notice that doublewarn is calling a nonexistent function.

Console

The interpreter and functions

When the interpreter encounters a function definition, it “remembers” the name of the function and how many parameters it takes.

- It does not go through the function body

When the function is called, the interpreter goes back to the function body and executes those commands

Variables in functions

- The function body can be thought of as a **separate** program.
- Variables created in the function “live” in a different area from other variables.
- In this separate ‘world’, variables in a function can even reuse variable names that were in the main program (but don’t!)

Example: variable scope

Next

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

When the program begins, it first encounters the function definition

Console

Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Next

It skips the body and goes to the next line

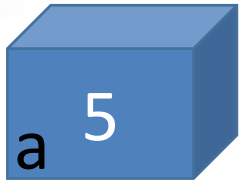
Console

Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Next

MAIN PROGRAM MEMORY



The variable a is defined in the main program's memory, and it is assigned the value 5

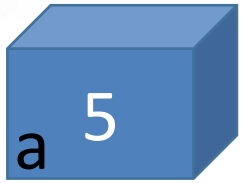
Console

Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Next

MAIN PROGRAM MEMORY



We print the value of a, which is 5

Console

5

Example: variable scope

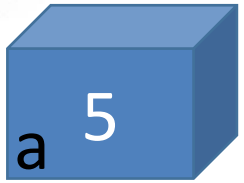
ENGINEERING
TEXAS A&M UNIVERSITY

Next

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Return HERE

MAIN PROGRAM MEMORY



my_function MEMORY

We now have a function call, so we go up to the function body.
When this happens, the function gets its own area of memory

Console

5

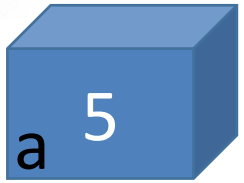
Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

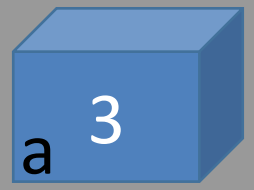
Next

Return HERE

MAIN PROGRAM MEMORY



my_function MEMORY



The variable a is defined in the function. This creates a new variable **in the memory area for that function**.
Notice that it is not the same as the earlier variable, a.

Console

5

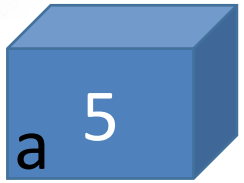
Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

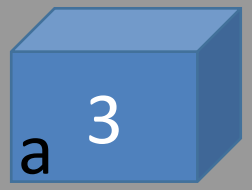
Almost Next

Return HERE

MAIN PROGRAM MEMORY



my_function MEMORY



When we print the value of a within our function, it will print the value in our function's memory. In this case, that's the number 3.

Console

5

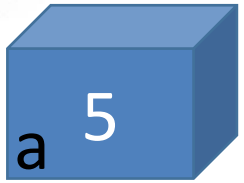
3

Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Next

MAIN PROGRAM MEMORY



When the function finishes, its memory is released. The variables created in that function are gone.

Console

5

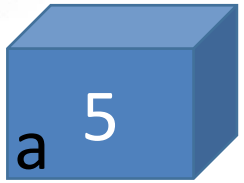
3

Example: variable scope

```
def my_function():
    a = 3
    print(a)

a=5
print(a)
my_function()
print(a)
```

Next

MAIN PROGRAM MEMORY



Printing the value of a prints the value in the main program's memory.

Console

5

3

5

Scope

The scope of a variable refers to the regions of the program where that variable is valid, and can be accessed.

Two terms are often used when discussing scope:

- A **local** variable is defined in the context of a particular function. Local variables are not shared between functions.
- A **global** variable is defined throughout the program - anything in the main code. This sometimes includes values imported from a module (e.g., pi)

Unless a variable with the same name has been defined locally in a function, the variable from the main code can be **read** in any function

(It usually can't be assigned to, though.)

What would this do?

```
def my_function():
    print(a)

a=5
print(a)
my_function()
print(a)
```

Notice that there is no longer variable a defined in the function.

Console

What would this do?

```
def my_function():
    print(a)

a=5
print(a)
my_function()
print(a)
```

This prints 5 in the function.

If the variable has not been defined in the function's memory, then it looks to the main program memory to get the value.

Console

5

5

5

What would this do?

```
def my_function():
    print(a)

a=5
print(a)
my_function()
print(a)
```

This is bad programming style, though!

Almost never access variables in the main memory from a function.

Functions should stand on their own – looking at only the function a person should understand what it does.

Console

5

5

5

Passing Parameters

The “right” way to access data from a function is to pass in the data via arguments/parameters.

In the function header, you list the variables your function will use.

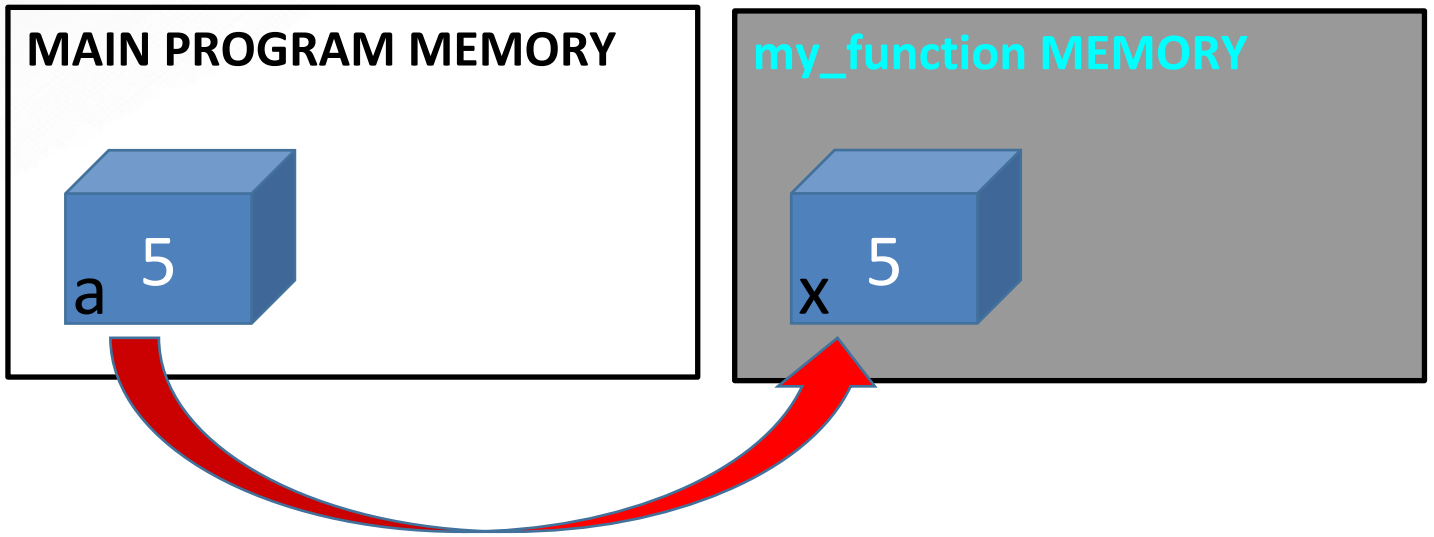
Example: `def my_function(param1, param2):`

- Creates a function that takes two parameters, called param1 and param2 in the function.
- When we call the function, we pass one argument per parameter:

```
my_function(3, "Joe")
```

Parameter Passing

```
def my_function(x):
    print(x)

a = 5
my_function(a)
```



When a function is called, the values of the arguments in the call are copied into the variables contained in the function's memory.

There is a new variable created in the function's memory space for each parameter.

The function uses the name of the parameter that it defines.

Example: passing parameters

ENGINEERING
TEXAS A&M UNIVERSITY

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

MAIN PROGRAM MEMORY

The interpreter noted the definition of my_function, but skipped over it for now.

Console

Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

MAIN PROGRAM MEMORY

Variable a is created in the main program.

Console

Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

MAIN PROGRAM MEMORY

The value of a is printed out.

Console

5

Example: passing parameters

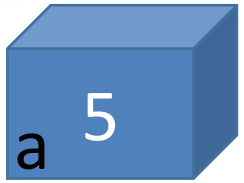
ENGINEERING
TEXAS A&M UNIVERSITY

Next

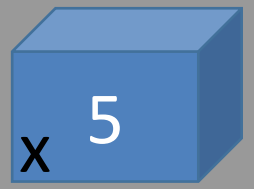
```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Return HERE

MAIN PROGRAM MEMORY



my_function MEMORY



The function call causes a new section of memory to be created. In that section, there is a variable named x created.
The value of the argument (5) is copied into this variable.

Console

5

Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

Return HERE

MAIN PROGRAM MEMORY

my_function MEMORY

The value of x is increased by 1.
The value of a is unchanged.

Console

5

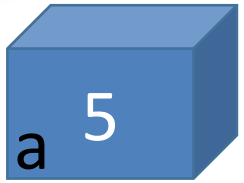
Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

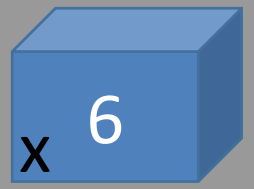
Next

Return HERE

MAIN PROGRAM MEMORY



my_function MEMORY



The value of x is printed.

Console

5

6

Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

MAIN PROGRAM MEMORY

The function completed so we returned to the place of the function call. The function memory is released.

Console

5

6

Example: passing parameters

```
def my_function(x):
    x += 1
    print(x)

a=5
print(a)
my_function(a)
print(a)
```

Next

MAIN PROGRAM MEMORY

We print the value of a.

Console

5

6

5

Exercise 1 – What would this output?

```
def F(x):
    x += 3

a = 5
F(a)
print(a)
```

Console

Exercise 2 – What would this output?

```
def F(a):
    a += 3

a = 5
F(a)
print(a)
```

Console

Exercise 3 – What would this output?

```
def F():
    a += 3

a = 5
F()
print(a)
```

Console

Exercise 4 – What would this output?

```
def F(a):
    print(a)

a = 5
b = 10
F(b)
```

Console

Declaring globals

Sometimes, you really do want to access some variable not passed in as a parameter.

- Generally, avoid doing this - but sometimes it really is the nicest thing to do

In the function, write a command: `global <variable name>`

The function now has full access to that global variable (for writing, as well as for reading).

Example: global variable

```
def F():
    global a
    a += 3

a = 5
F()
print(a)
```

The line “global a” means that the function now has access to the global variable, a, within itself. So, now you can change the value of the global variable within the function.

There are times this is very useful, for example to initialize a lot of different variables to values, but you should generally avoid doing it.

Console

8

Return values

Functions can also return values

- Always a single variable
- Use tuples to return multiple values

To return a value, use the command: `return <value>`

When this is encountered, the function *immediately* ends, returning the value specified.

Example 1: return values

```
def twenty_one():
    return 21

a = twenty_one()
print(a)
```

Console

Example 2: return values

```
def is_three_more(a, b):
    if a == b+3:
        return True
    else:
        return False

print(is_three_more(10,7))
print(is_three_more(1,5))
```

Console

Example 3: return values

```
def F(a):
    a += 1
    return a
    a += 10
    return a

x = F(4)
print(x)
```

Console

Default parameters

Sometimes we assume parameter values, but allow users to specify something different if they would like.

- This is assigning **default parameter values**
- Parameters are assigned default values with an = sign, followed by the value to take if the parameter is not specified.
- If the user does not specify a parameter, then the default is used.
- Default values must be the last input parameters

Example 1: default parameters

```
def F(a="Bryan", b="Texas", c="USA"):
    print(a, b, c)

F("Toronto", "Ontario", "Canada")
F("Orlando", "Florida")
F("Houston")
F()
```

Default values are assigned to all three parameters.

Notice that any parameters that are specified are filled in from left to right

Console

```
Toronto Ontario Canada
Orlando Florida USA
Houston Texas USA
Bryan Texas USA
```

Example 2: default parameters

```
def F(a, b="Texas", c="USA"):
    print(a, b, c)

F()
```

Parameters without a default value must be specified.

Console

```
TypeError: F() missing 1 required positional argument: 'a'
```

Example 3: default parameters

```
def F(a, b="Texas", c):
    print(a, b, c)
```

Once a default value is specified, all parameters after it in the parameter list must have defaults specified.

Console

```
SyntaxError: non-default argument follows default argument
```

Examples

Let’s look at a few more examples. We’ll write a function that takes in some parameters, modifies them, and then returns.

Example 1: function behaviors

```
def dosomething(a, b):
    a += b
    b = 7

x = 1
y = 2
dosomething(x,y)
print(x)
print(y)
```

Console

Example 2: function behaviors

```
def dosomething(a, b):
    a += b
    b = 7

x = 1.0
y = 2.0
dosomething(x,y)
print(x)
print(y)
```

Console

Example 3: function behaviors

```
def dosomething(a, b):
    a += b
    b = 7

x = "Texas"
y = "Aggies"
dosomething(x,y)
print(x)
print(y)
```

Console

Example 4: function behaviors

```
def dosomething(a, b):
    a += b
    b = 7

x = [1]
y = [2]
dosomething(x,y)
print(x)
print(y)
```

Console

Lists are Mutable

Lists are a mutable data type *(along with dicts, sets, and byte arrays)*.

You can think of things this way:

- Lists *cannot* be assigned as a brand-new list
- Lists *can* have their *elements* changed

Example: function behaviors

```
def dosomething(a):
    a = [10, 11, 12]

x = [1, 2, 3]
dosomething(x)
print(x)
```

Assigning a new value to a does not change the variable

Console

[1, 2, 3]

Example 1: function behaviors

```
def dosomething(a):
    a[0] = 10

x = [1, 2, 3]
dosomething(x)
print(x)
```

But, assigning a value to one of the **elements** of a does change that element.

In other words, the list can be changed, but it can't be reassigned.

Console

[10, 2, 3]

Changing Lists

So, for this reason, assigning to a local list variable will not change anything about the original list

- You'd be changing the place in memory where it is looking.

But, assigning to the **elements** of a local list variable **can** change the elements of the original list

- You're still accessing the same memory locations