Functions

15-110 – Wednesday 09/09

Learning Objectives

- Identify the **inputs**, **returned value**, and **side effects** of a function call
- Write new functions by identifying an algorithm's **steps**, **input**, **output**, and **side effects**
- Recognize the difference between local and global scope

Repeating Actions is Messy

Sometimes we want to perform the same algorithm many times on different inputs.

For example, say we want to personalize a young child's reading material so that it uses their pet's name.

We could copy and paste the first bit of code, then change the necessary parts. But if we're sloppy this might cause errors.

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

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!")
```

Functions Represent Abstract Actions

A better approach is to put the core action being repeated into a **function**.

A function is a code construct that represents an algorithm using abstraction. We can implement a function once, then **call** it many times.

Using Abstraction to Make a Function

To implement our problem as a Python function, we would write:

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

tellStory("Spot")
tellStory("Stella")
tellStory("Kimchee")
```

- What is this "def" thing?
- Why is there a colon at the end of the first line?
- Why are some lines indented?

It's **Python syntax**. All will be explained in this lecture.

Function Calls

To Call A Function, Use Parentheses

We've already seen how to call a function on a specific input, with built-in functions. This is done using **parentheses**.

```
functionName(input1, input2, ...)
```

The number of inputs provided inside the parentheses depends on how many inputs the function expects. Each input should be an **expression**.

Function Calls Follow Order of Operations

Function calls are **expressions**. They are evaluated and produce a value. That means we can **nest** them inside other expressions, the same way we can nest basic operations.

```
print(type(int("4") + float(3)))
```

Just as in math, functions follow the order of operations specified by the parentheses. Start by evaluating the inner-most expressions, `int("4")` and `float(3)`. Then evaluate the call to `type`; finally, evaluate the call to `print`.

Built-in Functions We've Already Seen

```
abs(-2)   # absolute value
type(2)   # the type of the value
str(2)    # converts the value to a string
int(3.5) # converts the value to an integer
float(5) # converts the value to floating point

print("Hello World") # displays the value
print(1, 2, 3) # print can take any number of inputs
```

Components of Function Calls

The functions we call may have three components:

Arguments – the values that are provided as input

Returned value – what the function evaluates to after running

Side effect – any change that happens while the function runs

Arguments are Inputs

The **arguments** we provide to a function are its inputs, just like the input to a generic algorithm.

Arguments are separated by commas and placed between the parentheses of the function call. Functions can require as many (or as few) arguments as needed.

The **positions** of the arguments usually have meaning. In the built-in function `pow(x, y)`, the first argument is the base and the second argument is the exponent. So `pow(2,3)` produces a different result than `pow(3,2)`.

The Returned Value is the Output

When a built-in function takes an input and runs through its algorithm, we cannot see what it is doing.

When the function is done, it sends back a result as a **returned value**. We usually say a function **returns** a value. This value substitutes in for the function call the same way a variable's value substitutes in for the variable itself.

For example, the returned value of `pow(2,3)` is `8`.

Side Effects Show Process

Sometimes a function has an observable effect in some way; for example, it might display a message in the interpreter, or modify a file, or produce graphics. This is called a **side effect**.

Side effects are different from returned values. A function might have no side effects, or one, or many. But every function has exactly one returned value.

Importantly, returned values can be saved and used in **future computations**. Side effects cannot.

The built-in function `print()` has a side effect: it prints to the screen. What is its return value?

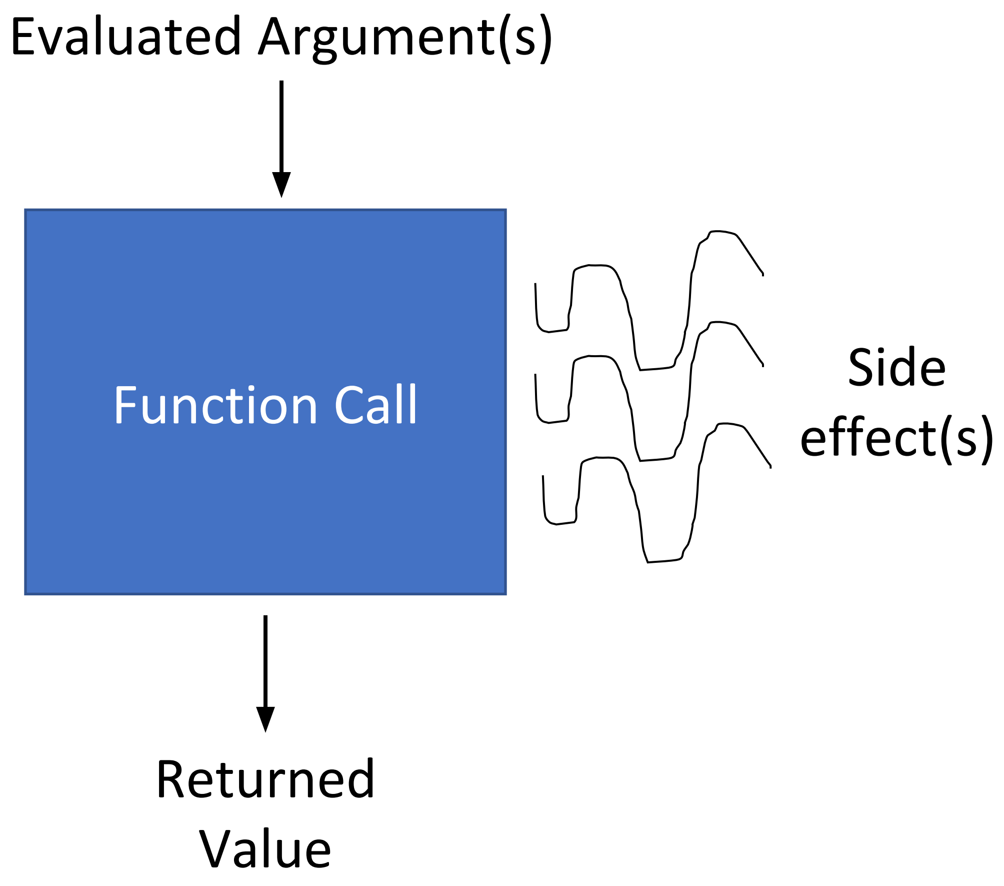
Missing Returned Values are None

If a function produces no explicit result (usually because it’s only used for its side effects, like print), it still has a returned value. That value is the built-in value None.

None means that there is nothing interesting to return. Like `True` and `False`, its meaning is built in to Python, so it does not need quotes.

If you try to set a variable to the result of `print()`, you'll find that the variable’s value is None. But the interpreter usually doesn’t display the value "None", it just says nothing. Try it!

Function Call Process



Activity – Identify the Function Call Parts

Consider the following two function calls. For each function call, what are its **arguments, returned value,** and **side effects**?

```
int(0.07)
```

```
print("15", "-", "110")
```

Import Adds Code from Libraries

The Python language has a ton of functions available for you to use, but most aren't included in the "built-in" package — the one you get by default. Instead they are organized into separate **libraries**.

To use a function from a library, you must **import** the library. This makes it possible to access the functions and variables in that collection.

All the Python libraries have **documentation** online that describes which functions are available and what they do. Find it by searching docs.python.org/3/

Importing the Math Library

We can import the **math** library to add more mathematical capabilities. Note that we must put `math.` in front of each function or variable name we use, to specify it came from that library.

```
import math
print(math.ceil(6.5)) # ceiling of a float number
print(math.log(64, 2)) # finds the log of 64 with base 2
print(math.radians(90)) # converts degrees to radians
print(math.pi)) # it's π!
```

Graphics as Side Effects

Graphics are Side Effects

Print statements technically create side effects, but this can be unclear when working in the interpreter because the printed text appears on the console, the same as return values. **Graphics** provide a clearer example.

To write code that produces graphical results (instead of only text results), we need to import a new library that lets us draw shapes on the screen.

```
import tkinter
```

Tkinter Starter Code

We'll need to run some code before and after our graphics code to make it work.

The `root` is the window. The `canvas` is the thing on the window where we can draw shapes.

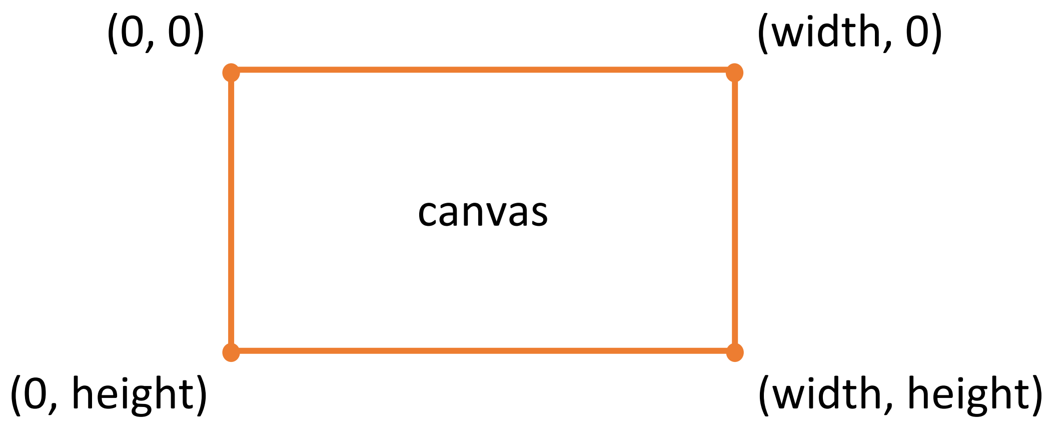
The last line will tell the window to stay open until we press the X button.

```
import tkinter

root = tkinter.Tk()
canvas = tkinter.Canvas(root,
             width=400, height=400)
canvas.pack()


# your code goes here


root.mainloop()
```

Coordinates on the Canvas Grow Down-Right

The **canvas** created by the starter code is the thing we'll draw graphics on. It's a two-dimensional grid of pixels. This grid has a pre-set **width** and **height;** the number of pixels from left to right and the number of pixels from top to bottom.

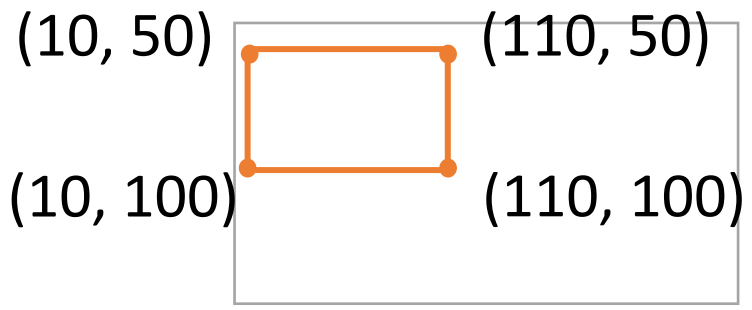
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on mathematical graphs – the origin starts at the **top left corner** of the canvas.



Drawing a Rectangle

To draw a rectangle, use the function `canvas.create_rectangle`. This function takes four required arguments: the x and y coordinates of the **left-top** corner, and the x and y coordinates of the **right-bottom** corner. The rectangle will then be drawn between those two points.

```
canvas.create_rectangle(10, 50, 110, 100)
```



Graphics – Side Effects and Returned Values

When the rectangle is drawn on the canvas, we can't access it or use it in future computations. It's purely a **side effect**.

But the graphics function call also has a returned value – an integer ID associated with the drawn shape. We won't use that value in this class.

You can draw a lot more than just rectangles with Tkinter graphics! Check out the bonus slides on graphics if you want to learn more.

Function Definitions

Writing Functions Takes Multiple Parts

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

To write a function, first determine the algorithm you want to implement, then identify the **steps**, **input**, **output**, and **side effects** of that algorithm.

Writing Functions – Body for Steps

Let's start with a simple function that has no 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.

When the indentation stops, the function is done. The last line **calls** the function we've written.

Writing Functions – Parameters for Input

If we want to add input to the function, we do so by adding **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 function arguments given each time the function is called.

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

hello("Scotty")
hello("Dippy")
```

Writing Functions – Return Specifies Output

To make our function have a non-None output, we need to add 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")
```

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

Writing Functions – Adding Side Effects

If we want to make sure our function performs a certain side effect, add that side effect to the function's body somewhere before the return statement. If the function returns None, no return statement is needed.

```
def drawDonut(canvas): # call drawDonut in graphics
    # create_oval makes an oval in the given bounding box
    canvas.create_oval(100, 100, 300, 300)
    canvas.create_oval(180, 180, 220, 220)
```

Example Side Effect Function

The following function has several side effects, and a return value of None.

```
def singHappyBirthday(name):
    print("Happy birthday to you")
    print("Happy birthday to you")
    print("Happy birthday dear " + name)
    print("Happy birthday to you!")

singHappyBirthday("Dippy")
```

Example Return Function

On the other hand, this function returns a float value and has no side effects.

```
def addTip(cost, percentToTip):
    return cost + cost * percentToTip

total = addTip(20.00, 0.17)
```

Activity: Side Effects and Returned Values

Now it's your turn! What are the side effects and returned values of the following function?

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

Scope

Variables Have Different Scopes

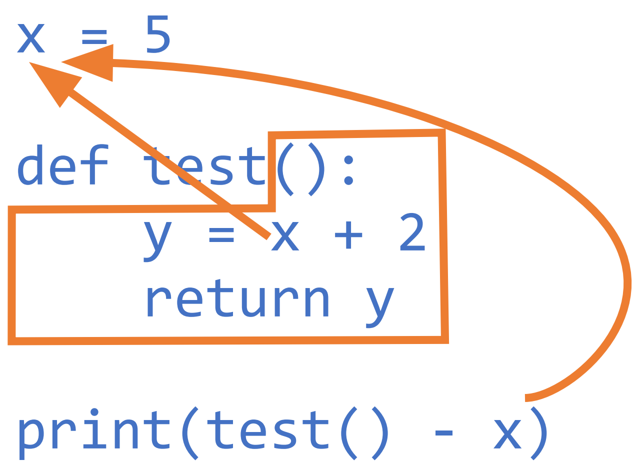
All the work done in a function is only accessible in that function. In other words, if we make a variable in a function, the outer program can't access it unless we return it.

```
def addItUp(x, y, z):
    answer = x + y
    answer = answer + z

print(answer) # NameError!
```

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) to see if the variable exists there.



```
x = 5

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

print(test() - x)
```

It's unlikely that you'll want to use this, but good to know for debugging.

Scope is Like A Person’s Last Name

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 x created inside the function would be *test*, while the last name of the top-level x would be *global*.

In general, it's best to keep the names different anyway, 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("Wednesday")
leave()
```

Sidebar: Python Analyzes the Entire Function

```
frog = "King Harold"

def greet1():
  print("Hello", frog)


greet1()
Hello King Harold
```

```
frog = "King Harold"

def greet2():
  frog = "Kermit"
  print("Hello", frog)


greet2()
Hello Kermit
```

```
frog = "King Harold"

def greet3():
  print("Hello", frog)
  frog = "Kermit"


Error: local variable
'frog' referenced
before assignment.
```

Learning Objectives

- Identify the **inputs**, **returned value**, and **side effects** of a function call
- Write new functions by identifying an algorithm's **input**, **output**, **steps**, and **side effects**
- Recognize the difference between local and global scope
- **Feedback: https://bit.ly/110-feedback**