Function Calls

15-110 – Wednesday 01/26

Announcements

- HW1 due Monday 01/31. Start early!
- Recitation tomorrow, still remote
- Grades for Check 1 are out
  - Can resubmit by revision deadline (02/08)
  - How revision deadline works

Learning Objectives

- Use **function calls** to run pre-built algorithms on specific inputs
- Identify the **argument(s)** and **returned value** of a function call
- Use **libraries** to import functions in categories like math, randomness, and graphics

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

We can also use functions that have already been defined by Python.

Function Calls

Call Functions with Parentheses

We've already seen how to call a function on a specific input, because `print` is just a function! 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**.

A Few New Functions

To help us explore how functions work, let's introduce a few new functions. These are **built-in functions**, like `print`; that means we can call them in Python directly.

```
abs(-2) # absolute value
pow(2, 3) # raises a number to the given power
round(12.4567, 2) # rounds to the given # sig digs
```

Type Functions

There are a few other built-in functions that are helpful to know, as they let you change the type of data values. This is called **type-casting**.

```
int("4") # converts a value to an integer
float(3) # converts a value to a float
str(98.9) # converts a value to a string
bool(0) # converts a value to a Boolean

type(4 + 3.0) # returns the type of the eventual value
# uses the names we covered before – int, float, str, bool
```

Components of Functions

The functions we call may have two core components:

Argument(s) – the values that are provided inside the parentheses, the **input**

Returned Value – what the function evaluates to after running, the **output**

Arguments Provide the Input

The specific inputs we provide to a function are called **arguments**. These are like the specific bread, peanut butter, and jelly we used in the PB&J algorithm. In the function call `abs(4)`, the argument is `4`.

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 `pow(2, 3)`, the first argument is the base and the second argument is the exponent. In other words, `pow(2, 3)` and `pow(3, 2)` mean two different things.

Receive Output as Returned Value

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

When the function is done, it sends back an output 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.

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

Function Calls Follow Order of Operations

Function calls evaluate to a single returned value; that means they are **expressions**. Therefore, we can **nest** function calls inside other expressions the same way we nest basic values and operations.

```
print(round(pow(abs(-12), 1/2), 2))
```

Just like in math, functions follow order of operations using parentheses. Start by evaluating the inner-most expressions, `abs(-12)` and `1/2`. Then evaluate the call to `pow`; then evaluate the call to `round`. Finally, evaluate the call to `print`.

Activity – Write Code Using Functions

You do: write a line of code in the interpreter that takes a variable `x` which holds a number as a string, turns it into an integer, and then doubles that integer.

For example, if `x = "21"`, then your line of code should produce `42`

Missing Returned Values are None

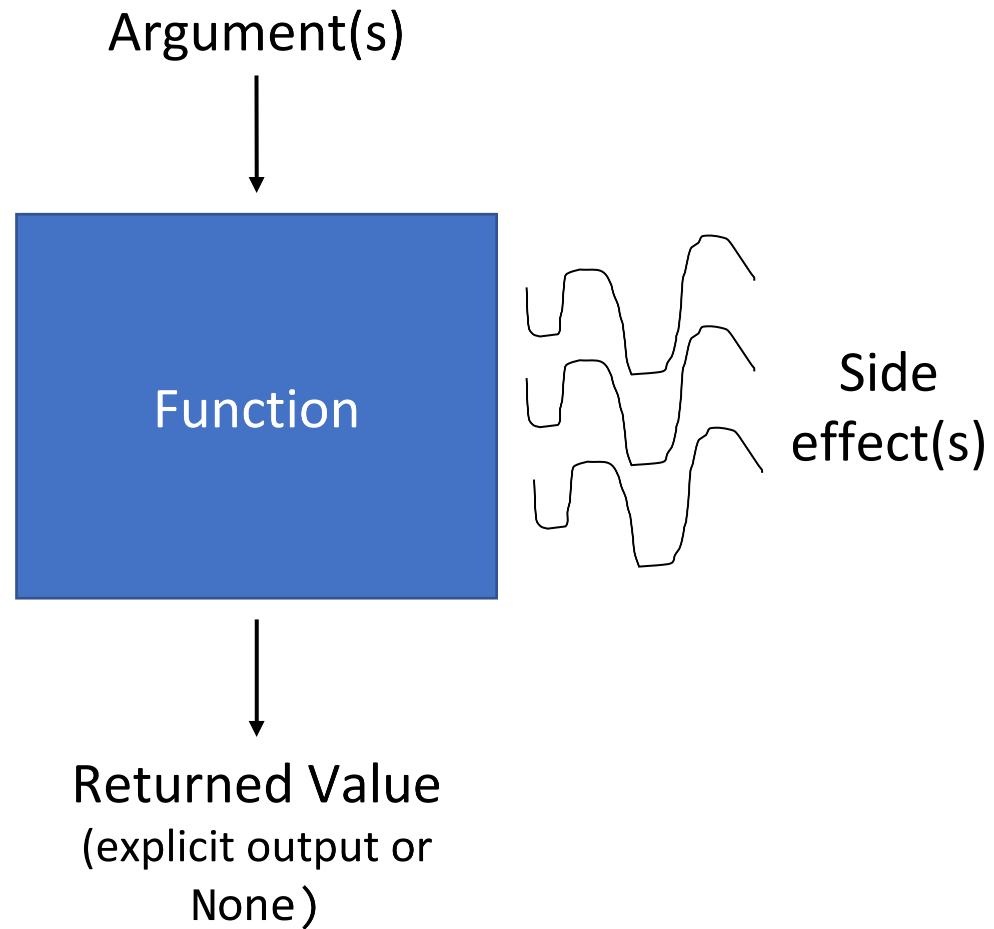
If a function produces no explicit output, it still has a returned value. That value is the built-in value `None`.

`None` means that there was no explicit output to be returned. Like `True` and `False`, its meaning is built into Python, so it does not need quotes.

If you try to set a variable to a `print` call, you'll find that the variable holds `None`. Note that `None` does not show up in the interpreter unless you explicitly `print` it; the interpreter just shows a blank instead.

```
>>> None

>>> print(None)
None
```

Function Call Process



Side Effects Show Change

If `print` doesn't have an explicit returned value, what exactly is it doing?

Recall that a program has a **state** that holds the current information that the program knows (what has been printed, what values do variables hold).

Function calls themselves are expressions, as they evaluate to a data value (the returned value). But sometimes a function changes the program state in an observable way as it is running; for example, it might display values in the interpreter, or modify a file, or produce graphics. This is called a **side effect**.

If we call `pow(2, 3)`, there is no observable side effect. But `print("Hello")` has an observable side effect: it prints `"Hello"` to the screen.

Side Effect(s) vs Returned Value

It's easy to get confused about whether something is a side effect or a returned value. Why are these two things different?

The way we've set up function calls means that there must be **exactly one output:** the returned value. A function call might have no side effects, or one, or many; however, every function call has one returned value.

Importantly, returned values can be saved in a variable and/or used in **future computations**. Side effects cannot be saved this way; we simply observe them.

Activity – Identify the Function Call Parts

Consider the following two function calls. For each function call, what are its **argument(s)** and **returned value?** Does it have any observable **side effect(s)**?

```
round(3.14159, 1)

print("15” + "-” + "110")
```

Libraries

Import Adds Code from Libraries

The Python language has a ton of pre-built functions, but most aren't included in the built-in package (the one available by default). Most of the functions 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. You can do this with the code:

```
import libraryName
```

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/ . It's better to check the documentation as needed than to try to memorize library functions.

Importing the math Library

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

Importing the random library

Importing libraries lets us get more creative with programming. For example, the **random** library lets us generate random numbers, which can help produce novel behavior.

```
import random
random.randint(1, 10) # picks a random int between 1-10 inclusive
random.random() # picks a random float between 0-1
```

Importing a graphics library

Finally, to get really creative, we can produce graphics with programming! We'll do this with the **tkinter** library, which makes it possible to draw shapes on a separate screen.

```
import tkinter
```

Tkinter Starter Code

We 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.

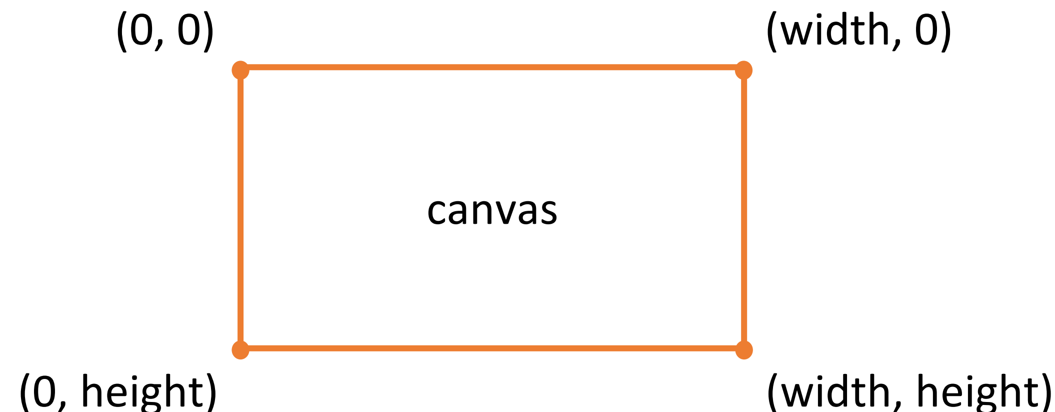
The `root.mainloop()` line will tell the window to stay open until we press the X button.

```
import tkinter

root = tkinter.Tk()
canvas = tkinter.Canvas(root,
                        height=400,
                        width=400)
canvas.configure(bd=0,
                 highlightthickness=0)
canvas.pack()

# write your code 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.

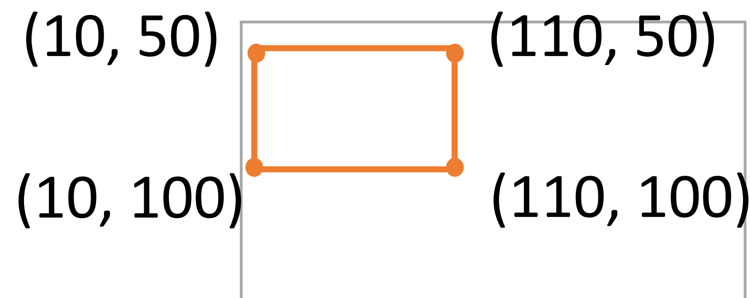
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on mathematical graphs – the origin starts at the **top left corner** of the canvas.



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)
```



Keyword Arguments Add Variety

With the basic parameters, we can only draw outlines of shapes. By adding **keyword arguments**, we can change the properties of these shapes.

A keyword argument is an argument that is associated with a specific name instead of a position in the function call. We can put keyword arguments in any order we like as long as they occur after the main arguments.

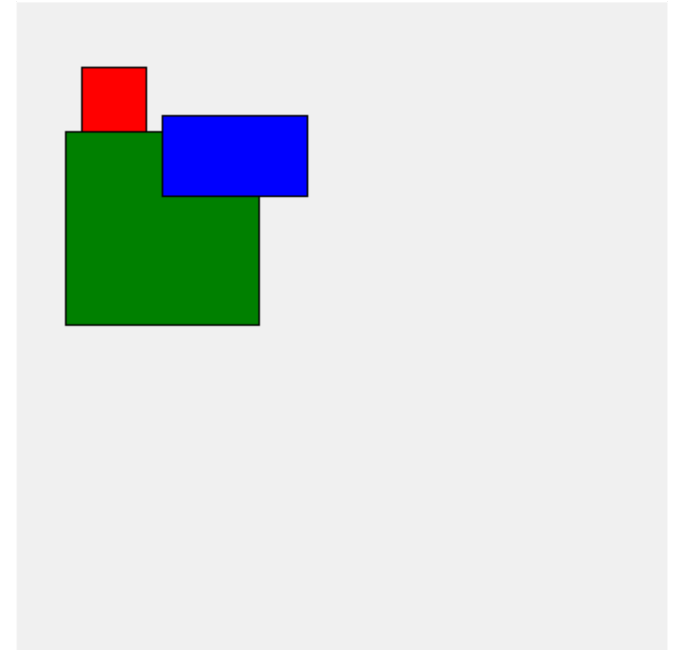
Keyword arguments can have **default values**, which is why we don't need to include them in every graphics call. To change that default value, include the keyword, followed by `=`, followed by the new value in the function call.

```
canvas.create_rectangle(50, 100, 150, 200, fill="green")
```

Keyword Argument - fill

The `fill` argument can be used to give a rectangle a color. It uses a string (the name of the color) to change the color of the shape.

```
canvas.create_rectangle(40, 40, 80, 140, fill="red")

canvas.create_rectangle(30, 80, 30 + 120, 80 + 120,
                        fill="green")

canvas.create_rectangle(90, 70, 180, 120, fill="blue")
```



Note that when we draw shapes on top of each other, the one on top is the **last one called**. Order matters!

Graphics – Side Effects and Returned Values

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

The graphics function call also returns something – 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 to find more shapes and keyword arguments.

Activity: Try it out!

You do: try running a function call in the graphics starter code to generate a shape or two. Then try applying the keyword argument `fill` to change the shape's color.

Learning Objectives

- Use **function calls** to run pre-built algorithms on specific inputs
- Identify the **argument(s)** and **returned value** of a function call
- Use **libraries** to import functions in categories like math, randomness, and graphics

Feedback: https://bit.ly/110-s22-feedback