#4: Functions and Scope

SAMS SENIOR NON-CS TRACK

Last Time

Use **variables** to hold and update data

Use **input and output** to write code that supports user interaction

Ex3-1 Feedback

Excellent work- everyone got most of the problems right!

Today's Learning Goals

Use **functions** to hold and execute processes

Understand how **scope** changes where we can access variables

Functions

Functions are Processes

We've now used variables as a way to store **data** so that it can be used several times. Functions are similar to variables, except that we'll use them to store **processes** so that they can be used multiple times.

We'll define a process to be a **sequence of statements** that have some effect. Often, we'll only be able to see the end result of the process, even if more work is being done behind the scenes.

Built-in Functions

We'll start by considering functions that Python has already implemented. These are automatically available whenever you run Python.

We can **call** a built-in function by using the function's **name**, then following it with **parentheses**. We'll provide any values the function needs for its computations inside the parentheses. We call these values **arguments**.

We've already used a few built-in functions in the class, as is shown to the right.

```
print("Hello World")
canvas.create_rectangle(0, 0, 400, 400)
input("What's your name?")
str(56) # changes integers to strings
int("4") # changes strings to integers
```

Note that some functions (print and canvas.create_rectangle) make a change, while others (str and int) produce a value that can be used directly.

More Built-in Functions

There are plenty of other built-in functions in Python that we can use! This includes:

```
len("Hello World") # evaluates to the length of the given string
float("6.591") # changes the string to a decimal number (floating-point)
max(4, 6, 2) # evaluates to the maximum number provided as an argument
min(3, 9, -5) # evaluates to the minimum number provided as an argument
round(3.14159, 2) # evaluates to the first number rounded to
# the second number of digits
```

For a full list, check out https://docs.python.org/3/library/functions.html

Exercise 1: Rounding

Go to the schedule page and download the starter file for today's lecture. You'll write exercise code under the comment with the exercise's number.

Exercise 1: Write one or more lines of code that interactively asks the user to enter a number, then prints out that number rounded to one decimal place.

For example, if I entered "4.234", it would print 4.2; if I entered "6.591", it would print 6.6

Defining a Function

We're not limited to the functions that Python has already defined- in fact, we can define our own functions, and then try calling them!

At the most basic level, a function needs to have a **name** and a **body**. The name is how we will refer to the function; the body is the process that the function performs. Here's an example:

```
def <functionName>():
    <functionBody>
```

Indentation

Unlike previous code we've written, a single function needs to be defined **across multiple lines**. This can be just two lines (if the body is only one lines), or much more! How can Python keep track of what should be in the function and what should not?

We designate which lines should be in the function's body by **indenting** them (putting whitespace before the start of the text). In general, Python uses indentation to specify when one or more lines of code should be considered a 'block'. We'll need to use the same amount of leading whitespace for all the lines of the block. We often use a tab, or four spaces.

```
def greetPerson(): # Note that each line starts with four spaces
    name = input("What is your name?")
    print("Hello, " + name + "!")
print("Welcome to class.") # This line is outside the function, not part of the process
# We call code that happens outside of functions top-level code.
```

Calling our function

Once we have defined a function in a file, we can call it to run the code inside the function. To do this, we just need to refer to the name of the function, then put parentheses after it.

```
def greetPerson():
    name = input("What is your name?")
    print("Hello, " + name + "!")
print("Welcome to class.")
greetPerson() # We're calling the function here!
greetPerson() # And call it again here!
```

Exercise 2: Hello World Redux

Exercise 2: write a function called helloWorld which takes no input and simply prints the string "Hello, World!". Then call this function three times at the top-level of the code, so that "Hello, World!" is printed three times overall.

Parameters hold Input Values

Sometimes we want to provide additional information to a function that will change what it does (like how str() takes in information about the number it should change). To do this, we need to add **parameters** to the function definition.

Parameters are just variable names, but instead of being given starting values, they're put into the parentheses after the function name. These parameters will be assigned values **when we call the function**, and not before.

```
def <functionName>(<parameters>):
    <functionBody>
```

Note that we can have 0 parameters [()], 1 parameter [(param1)], 2 parameters [(param1, param2)], etc...

Parameter Example

If we use parameters, we can change our greeting program from before to take a name as a parameter instead of using input. Note that we now provide the name directly when we **call** the function. The function itself needs to use the abstract idea of a name instead of a direct value!

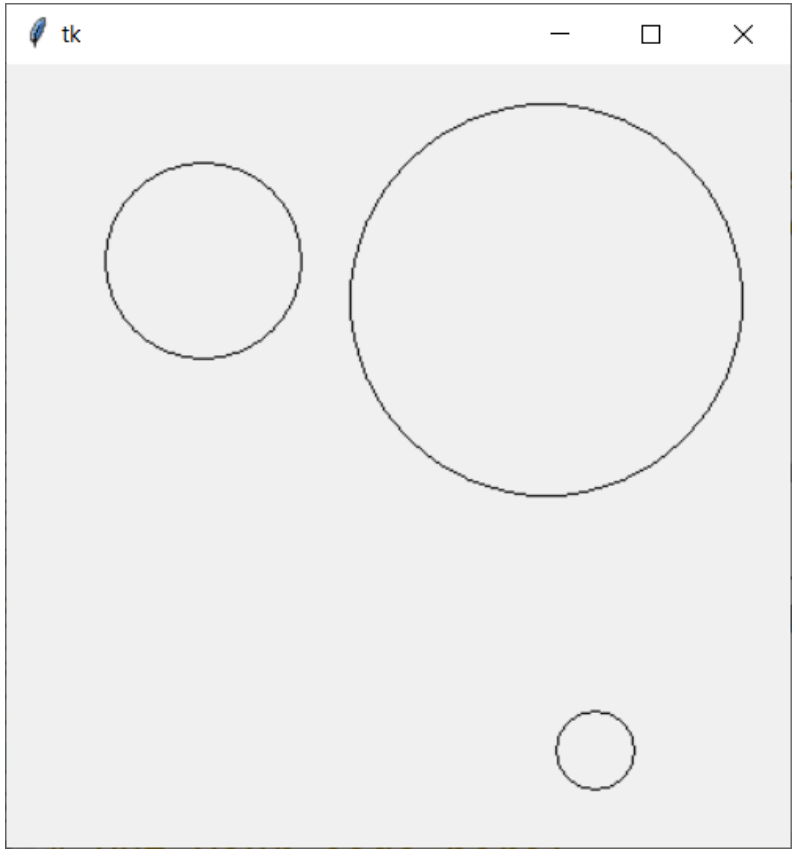
```
def greetPerson(name):
    print("Hello, " + name + "!")


print("Welcome to class.")
greetPerson("Gloria") # Here, name is set to "Gloria"
greetPerson("Denise") # And here it's set to "Denise"
```

Exercise 3: drawCircle

Exercise 3: write the function drawCircle(canvas, x, y, r). This function takes four parameters – the canvas, an (x,y) coordinate, and an integer r – and uses them to draw a circle centered at (x,y) with radius r.

Then call drawCircle at the top-level of the code with three different coordinates (and three different radii) to produce a picture with three circles in different locations on the canvas. An example is shown to the right (you don't need to mimic this exactly!).



Return Provides an Output Value

Finally, sometimes we want to make our function produce a specific value that we can then use in expressions later on (like how int('4') produces a 4). We call this **returning** a value from a function, and we do this by including a **return statement** in the function body.

When we run a return statement, it takes the value next to it and sends it back to the location where the function was called originally. It will then substitute in that value in the place where the function was called.

```
def <functionName>(<parameters>):
    <functionBody>
    return <functionResult>
```

As soon as we return, the function will end, so for now we always want the return statement to be the last statement in the body. Also, note that we can't write a return statement outside of a function!

Using Returned Values

Adding a return statement lets us write functions that will produce output, so that we can use that output in calculations. For example, we can write a function that will help us compute how large to draw a square if we want it to fill the canvas with a certain size of margin.

```
def squareSize(windowSize, margin):
    return windowSize - 2 * margin


canvas.create_rectangle(10, 10, 10 + squareSize(width, 10),
                                10 + squareSize(height, 10))
```

Note that I can call squareSize on (width, 10) assuming that width has already been defined. When we evaluate 10 + squareSize(width, 10), the function call will run, produce a result, then add that result to 10.

Another Type: None

What happens if we don't put a return statement in a function? The function will then return nothing- or, more specifically, the **None value**.

None is a unique built-in value. We can use it to tell the user that there is no returned result from the function

```
def greeter(name):
    print("Hello, " + name + "!")

print("Result:", greeter("Kelly"))
```

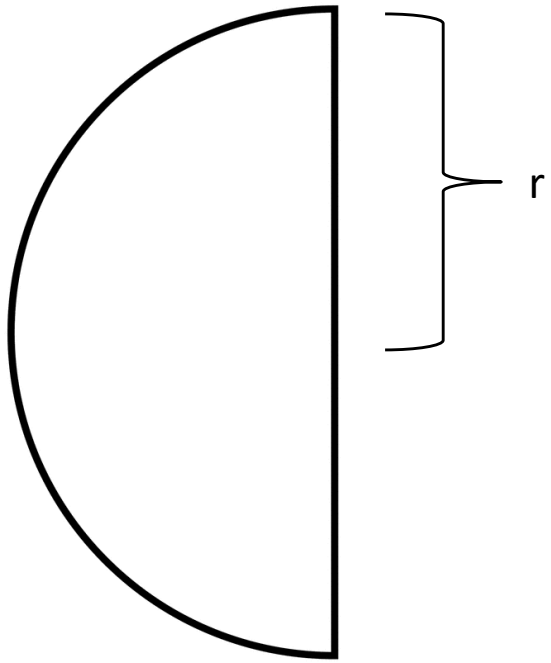
Note that the first printed value is the message printed by greeter(), but the second is Result: None, since greeter() did not actually return a value.

Exercise 4: Semicircle Perimeter

Exercise 4: write the function circlePerimeter(r) which takes a radius (r) and returns the perimeter of a circle with that radius. Recall that we calculate a circle's perimeter with 2πr. (You can use **math.pi** to get the value of π).

Then, at the top level of the code, write a line of code that prints the perimeter of a **semicircle** with a radius of 20. You must call your circlePerimeter() function to get full credit!

Hint: note that a semicircle has half the perimeter of a circle, plus a flat line that is equal to twice the radius.



Defining a Function: Example

In general, when we create a function, we want to identify an appropriate **identifier, input, output,** and **process** for that function. These values will directly translate to the function's **name, parameters, return value,** and **body**.

Say we want to define a function that converts money into a number of quarters. Our function components are:

Name: `convertToQuarters`

Parameter: `money`

Body: `numQuarters = money / 0.25`

Result: `return numQuarters`

```
def convertToQuarters(money):
    numQuarters = money / 0.25
    return numQuarters
```

Scope

Functions have a different scope

When we define variables inside a function, they **only exist inside the function**. We can't call them in the main code body.

Example:

```
def convertToQuarters(money):
    numQuarters = money / 0.25
    return numQuarters


print(numQuarters * 4) # will crash
```

Scope Organizes Names

This happens because Python considers function bodies to be in a different **scope** that the top-level code. We can only access variables in the scope in which they are defined.

One way to think about this is that a variable's name is its **first** name, but it's function name (or the top level) is its **last** name. You might have the same first name as another person at your school, but you probably have a different last name, and that helps to distinguish between the two of you.

In the example to the right, note that when we print x at the end, it doesn't change to 9 or 11. This is because x top-level is separate from x foo.

```
def foo(x): # This is x foo
    x = x + 2
    return x


x = 5 # This is x top-level


print(f(9))
print(x)
```

You Do: Code Tracing

What will the code to the right print out when we run it?

Try predicting the answer by writing out the steps on paper.

```
def a(x):
    y = 5
    return x + y
def b(x, y):
    return x - y
x = 10
print(a(x) + b(9, 4))
print(x)
print(y)
```

Functions Can Call Functions

We're not restricted to calling functions only at the top-level- we can also **call functions inside of other functions!**

```
def a(x):
    return x * 2
def b(y):
    return a(y) - 1
print(b(10))
```

This lets us write special functions that we'll call **helper functions**. We'll use these when we need to solve large or complicated problems, to break up the work into multiple parts.

The Stack Tracks Function Calls

When a program is calling multiple functions, how do we keep track of which function we're currently in and where the return values should be sent?

Python keeps track of something called a **stack**, which is basically a list of all of the places in the code where we need to eventually return a value. When we reach a return statement, Python removes the current value of the stack and goes back to the previous one.

In the following example, when we've reached line 2, the stack would look like this:

```
1: def a(x):
2:     return x * 2
3: def b(y):
4:     return a(y+1) - 1
5: print(b(10))
```

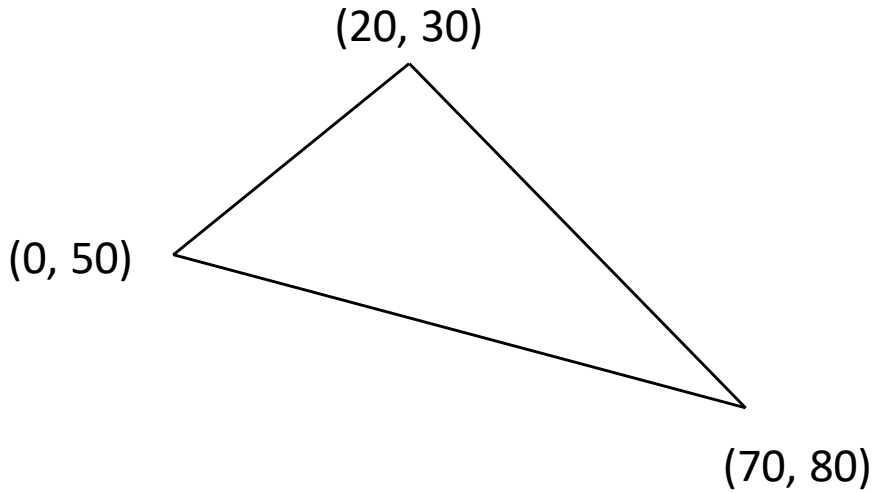
Line 5 – called b() on 10
Line 4 – called a() on 11
Line 2 – return 11 * 2 to previous item in stack

Exercise 5: trianglePerimeter

Exercise 5: write the function trianglePerimeter(x1, y1, x2, y2, x3, y3) which takes three coordinates – (x1, y1), (x2, y2), and (x3, y3) – and calculates the perimeter of the triangle made by connecting those three points.

To make solving this problem easier, you should also write the function distance(x1, y1, x2, y2) which takes two coordinates – (x1, y1) and (x2, y2) – and calculates the distance between them. You should then call distance() from trianglePerimeter().

Finally, print out the result of calling trianglePerimeter on the points (0, 50), (20, 30), and (70, 80) to find the perimeter of that triangle!



Today's Learning Goals

Use **functions** to hold and execute processes

Understand how **scope** changes where we can access variables