CSCI 141 - Spring 2019
Lab 5: Functions: Turtle Shape Functions
Due Date: Check Canvas

Introduction

This lab introduces you to the art of writing your own functions, the concept of local variables and parameters, and the importance of documenting the behavior of the functions you write. You will write a handful of functions that draw various shapes using a turtle, and use them to make complex drawings with ease.

1 Functions

This section of the handout reviews the basics of functions. Please read through it before getting started, and refer back to it if you encounter confusion or unfamiliar terminology when completing the activity. If you have any questions, your TA is there to help.

Basics

As we've seen in lecture, functions provide a way to assign a name to a given procedure, or a sequence of statements. We've been using (calling) functions that have been written for us since early in the course, such as

```
print("Hello, world!")
```

A function can take zero or more input arguments, have effects, and optionally return a value. For example,

```
  print("Hello world!")
```

takes one or more arguments as input, has the effect of printing them to the screen, and does not return a value.

Writing your own functions is an extremely powerful ability, because it gives you a way to create customizable pieces of code that you can then use as building blocks for creating more complicated programs. Recall from lecture that the syntax for declaring a function looks something like this:

```
def function_name(arg1, arg2):
    """An example function that demonstrates the syntax for
       writing your own functions."""
    statement_1
    statement_2
    statement_3
    return a_value
```

Here's an example program that declares a function that takes two inputs, computes their sum, and then prints the sum before returning it. The program then declares two variables and calls the function to print and return their sum.

```
# example of a function definition:
def calc_sum(num_1, num_2):
    """ Print, then return, the sum of num_1 and num_2
        Precondition: num_1 and num_2 are numbers. """
    the_sum = num_1 + num_2
    print(the_sum)
    return the_sum

a = 4
b = 6

# example call to the function:
ab_sum = calc_sum(a, b)
```

Notice that printing a value and returning it are two different things: printing is an effect that causes it to show up on the screen. Returning it means that the expression `calc_sum(num_1, num_2)` *evaluates to* the resulting sum. This means when we execute the assignment statement after the function definition, the returned value pointed at by `the_sum` will get assigned to the variable `ab_sum`.

Triple-Quoted Strings

Notice that the lines directly after after the function header (the line with the `def` keyword) contain a **triple-quoted string**. Enclosing a string in triple quotes is like using single or double quotes, except that newlines are allowed in triple-quoted strings.

```
my_str = "a normal string
with a newline" # will cause an error
triple_quoted_string = """A string in
triple-quotes""" # will not cause an error

print(triple_quoted_string)
# would print the following:
# A string in
# triple-quotes
```

Otherwise, a triple-quoted string behaves just like any other string.

Specifications

When writing functions in any language, it's standard practice to write a **specification** (or **spec**, for short): a description of what someone needs to know to use it. In particular, the spec typically includes a description of any parameters, effects, and the return value, if any, of the function. This makes it possible for other programmers to make use of your function without having to read through the function's code to figure out how it works. This is what you've been doing all along when calling functions like `print`: you know what it does, but you don't know how the code is written to accomplish its behavior.

Docstrings

In Python, the convention is to write function specifications in **docstrings**: triple-quoted strings that appear just after the function header (the line with the `def` keyword). The triple-quoted string is not technically a comment: it's actual Python syntax; but an expression (or a value) on a line by itself has no effect when executed by the Python interpreter, so it's presence doesn't change the behavior of the program. Consequently, it is not a syntactic requirement of the language to include a docstring in every function. However, it **is** a requirement of this course

to include a docstring in every function you write, unless you are told otherwise. Take a look at the docstring in `calc_sum` for an example.

What information should you include in your docstrings? Generally speaking, a programmer should know the following after reading your function's specification:

- The meaning of each parameter that the function takes
- Any effects that the function has
- The return value (and its type), if any, of the function
- Any **preconditions**: Assumptions that your function makes about the state of the program, or about the arguments passed to it.
- Any **postconditions**: Assumptions that can be made once the function call is finished about the program state or return value.

Note that it's a good idea to keep specifications as concise as possible. For example, the spec for `calc_sum` does not specify as a postcondition that the return value is the sum of the two inputs, because that's already stated in the description. However, it is worth including the precondition that the arguments are numerical, because otherwise an error might result. A user of your function is now aware that they shouldn't call `calc_sum` with non-numeric arguments. If they do and an error occurs, it's their mistake, not yours!

Local Variables

Notice that in the definition of `calc_sum`, we created a new variable called `the_sum`[1]. Because it was created inside a function definition, `the_sum` is what is known as a **local variable**, meaning that it doesn't exist (or isn't "visible") anywhere in the program except inside the `calc_sum` function definition. A variable's **scope** refers to the set of statements in a program where it is accessible; in this case, `the_sum`'s scope is within the `calc_sum` function. If we tried to refer to `the_sum` outside that indented code block, we'd get an error.

Variables such as `a`, `b`, and `ab_sum` are called **global variables** because once they are defined using an assignment statement, they are visible for the entire remainder of the program.

[1] I didn't call it `sum` because that's already the name of a builtin function; it's syntactically valid to create a variable with that name, but it "hides" the builtin function so you can't use it anymore because `sum` now refers to a varaible.

Parameters Are Local Variables

When defining a function, we need a way to refer to the arguments that are passed in when it's called. **Parameters** serve this purpose: in the `calc_sum` function definition, `num_1` and `num_2` are the function's parameters. When the function is being executed, `num_1` points to the value of the first argument and `num_2` points to the value of the second one. Consequently, parameters are simply special local variables that are automatically assigned the values passed to the function as arguments. Like any other local variable, their scope is limited to the function definition to which they belong. Referring to `num_1` or `num_2` outside of the function definition will result in an error for the same reason that trying to refer to `the_sum` will cause an error.

2 Shape Functions for Turtles

You'll now write some functions that will make it easier to make complicated drawings using Python's `turtle` module. For now, you'll be provided with the function header and specification, and it is your job to make sure that the function implements (or adheres to) the spec exactly and in all cases.

2.1 Setup

Create a `lab5` directory in your lab environment of choice. Fire up Thonny, create a new file, and save it as `turtleshape.py`. Write a comment at the top with author, date, and a description of the program. Download `turtleshape_test.py` from the course webpage and save it in your `lab5` directory alongside `turtleshape.py`.

2.2 Testing

One of the many benefits of writing small, self-contained functions is the ability to test them independently of other code that uses them. Once you've tested a function thoroughly, you can safely use it without fear of a bug lurking somewhere inside. It's a good idea to test functions one at a time, as you write them. Start by making calls to your function in the interactive shell (the bottom pane in Thonny) to see if they're working correctly.

Once you believe a function behaves as specified, open up `turtleshape_test.py`, and look toward the bottom for a commented-out call to a function named `test_`*function_name*. For example, for `draw_square`, the corresponding function is called `test_draw_square`. Remove

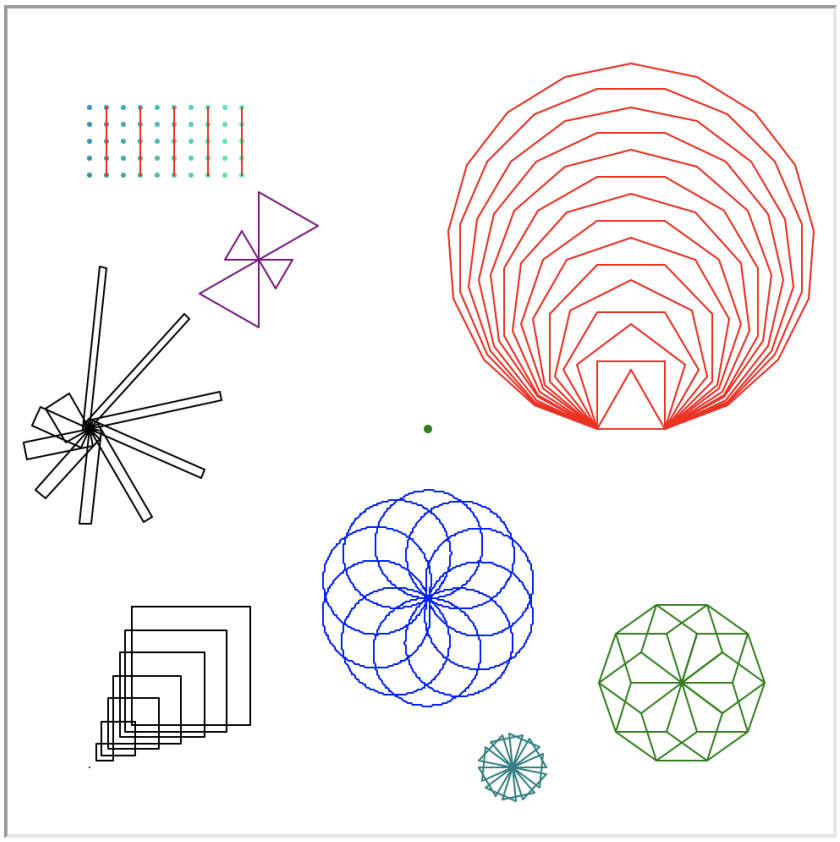

Figure 1: The correct output of `turtleshape_test.py` after all functions are completed.

the `#` from the beginning of the line to enable the test function, then hit the green Run button to run `turtleshape_test.py`. Each of the functions draws one piece of the drawing shown in Figure 1. For example, the black squares of increasing size in the bottom left should appear exactly as in the figure once your `draw_square` function works correctly.

To make sure that everything's set up correctly, first run the test program unmodified: you should see the drawing with only the green dot in the middle.

2.3 draw_square

In last week's lab, you wrote a loop to make a turtle draw a square. Now, let's do the same thing, but wrap it in a `draw_square` function so we can draw a square with a simple function call.

Header and Specification:

```
def draw_square(t, side_length):
    """ Use the turtle t to draw a square with side_length.
        Precondition: t's pen is down
        Postcondition: t's position and orientation are the same as before
    """
```

Type the function header and specification (docstring) into `turtleshape.py`, then write code in the function body to make the turtle draw a square.

Try out your `draw_square` function in the interactive pane, something like this:

```
>>> import turtle
>>> scott = turtle.Turtle()
>>> draw_square(scott, 100)
```

2.4 draw_rectangle

Next, write a more general function to draw a rectangle of any size. Notice that once we have a rectangle function, we can use it to draw a square by calling the rectangle function with equal side lengths.

```
def draw_rectangle(t, width, height):
    """ Draw a rectangle using turtle t with size width x height
        Precondition: t's pen is down
        Postcondition: t's position and  orientation are the same as before
    """
```

After uncommenting `test_draw_rectangle` in `turtleshape_test.py`, the orange spiral of rectangles should appear as in Figure 1.

2.5 draw_triangle

Another way to generalize the square function would be to draw different equilateral polygons (i.e., shapes with different numbers of equal side lengths). To get started, implement a `draw_triangle` function that draws a triangle with equal length sides:

```
def draw_triangle(t, side_length):
    """ Draw an equilateral triangle using turtle t with side_length
        Precondition: t's pen is down
        Postcondition: t's position and orientation are the same as before
    """
```

When completed and the corresponding test function is uncommented, the purple bowtie-like figure should appear as in Figure 1.

2.6 draw_polygon

You've now figured out how to draw a square (4-sided polygon) and a triangle (3-sided polygon). Now, write a function that draws an n-sided polygon:

```
def draw_polygon(t, side_length, num_sides):
    """ Draw a polygon with num_sides sides, each with length side_length
        using turtle t
        Precondition: t's pen is down; num_sides > 2
        Postcondition: t's position and orientation are the same as before
    """
```

The red pattern with nested n-gons should appear as in the Figure when this function works correctly.

2.7 draw_snowflake

One of the reasons that functions are so powerful is that we can **compose** them; in other words, one function can call another. Now that we have a function that draws polygons, it's pretty simple to make a function that uses it to create variations on the snowflake-like pattern from last week's lab:

```
def draw_snowflake(t, side_length, num_sides):
    """ Use t to draw a snowflake made of ngon-sided polygons. The snowflake
        contains 10 copies of a polygon with num_sides and side_length, each
        drawn at a 36-degree angle from the previous one.
        Postcondition: t's position and orientation are the same as before
    """
```

The teal, green, and blue snowflakes in the bottom right corner should appear as in the Figure once this function is working correctly.

2.8 `teleport`

Finally, write a function that implements a convenient ability: teleport the turtle to a given location without drawing, but leaving the pen state unchanged afterwards. This is similar to the `turtle` object's `goto` method, except it never draws. To accomplish this, you'll need to pick up the pen first, then put it down *only* if it started out down. You may find it helpful to look at the turtle documentation for methods that could be useful here.

```
def teleport(t, x, y):
    """ Move the turtle to (x, y), ensuring that nothing is drawn along the
        way. Postcondition: the turtle's orientation and pen up/down state is
        the same as before.
    """
```

When basic movement is working, the gradient-colored grid of dots should appear. When the pen is correctly restored to its previous state, the vertical red lines should appear as in the Figure.

2.9 Screenshot

Take a screenshot of the Turtle window when you have everything working, and name the file `turtleshape.png`.

3 Drawing

Your final task will be to write a short program that uses your drawing functions to make some interesting drawing.

Importing Local Files; Writing "Main" Functions

You may have noticed that the code in `turtleshape_test.py` calls functions from `turtleshape.py`. To make this possible, `turtleshape_test.py` had to execute `import turtleshape`; this is just like importing a module (like `math` or `turtle`, except the module is located right in the same directory. Python looks first in the local directory for a module with name `turtleshape.py`, then it imports the code if it is found.

What happens when importing code? Basically, it's like pasting the imported module's code into your file: it all gets run. So far, your `turtleshape.py` contains only function definitions, so importing it simply defines those functions. But if you wrote code outside the functions, it would get executed when you `import turtleshape`. Often, we want to separate the behavior of a file as a **program** versus as a **module**, so importing it causes functions to be defined but doesn't run the program, but you can still run the program, e.g., with `python turtleshape.py` or by clicking the Run button.

The way to do this is to use a so-called "main function" or "main guard". `turtleshape_test.py` contains an example of this. Basically, any code you want to run as a program but don't want to execute when you import your file as a module, you can put inside the following if statement:

```
if __name__ == "__main__":
  # code here will not run when this file is imported
  # but will run if the file is run as a program
  print("Hello, world!")
```

You don't need to worry about the details of why this happens, but it's a good thing to remember how to do. You can always google it if you forget the syntax.

A common way to use this is to have one function that contains your program code, usually called `main()`, and place a single call to that function inside the main guard:

```
def other_function():
    """ Return the number 4 """
    return 4

def main():
    """ Main program: print hello, world! """
    print("Hello, world!")

if __name__ == "__main__":
    main()
```

If this were in a file called `prog.py`, then running `python prog.py` would print "Hello, world!", whereas executing `import prog` would make `other_function` and `main` available for use, but wouldn't execute the call to `main()`.

Make a Drawing

Below all your function definitions in `turtleshape.py`, write some code that creates a drawing. Put your code in a main function and call it inside a main guard as illustrated above, so that running the test program (which `import`s your code) does not cause your main function to be called.

Your code should use at least one loop and make use of at least two of the functions you wrote in this lab. Feel free to also use other functions from the `turtle` module. Take a screenshot of your drawing and save it as `drawing.png`. Your drawing should complete in under a few seconds (use `turtle.tracer(0,0)` and `turtle.update()` as in Lab 4), and should match your screenshot.

Submission

At this point, show the output of the test program and your drawing to your TA so you can be immediately awarded points for it. Unless you do not finish during the lab period, do not leave until your TA has verified that your output is correct.

Create a zip file called `lab5.zip` containing `turtleshape.py`, `turtleshape.png`, and `drawing.png`. Upload your zip file to the Lab 5 assignment on Canvas. Even if your TA has checked you off, you still need to submit to Canvas.

Rubric

You submitted a single zip file called `lab5.zip`, containing the correct files	1
The top of your program has comments including your name, date, and a short description of the program's purpose. Comments placed throughout the code explain what the code is doing.	1
`draw_square` works correctly	3
`draw_triangle` works correctly	3
`draw_rectangle` works correctly	3
`draw_polygon` works correctly	3
`draw_snowflake` works correctly	3
`teleport` works correctly	3
Your drawing code is is inside a main guard	2
Your drawing code uses at least one loop and two of the functions you wrote.	6
Your drawing code runs in under a few seconds.	2
Total	30 points