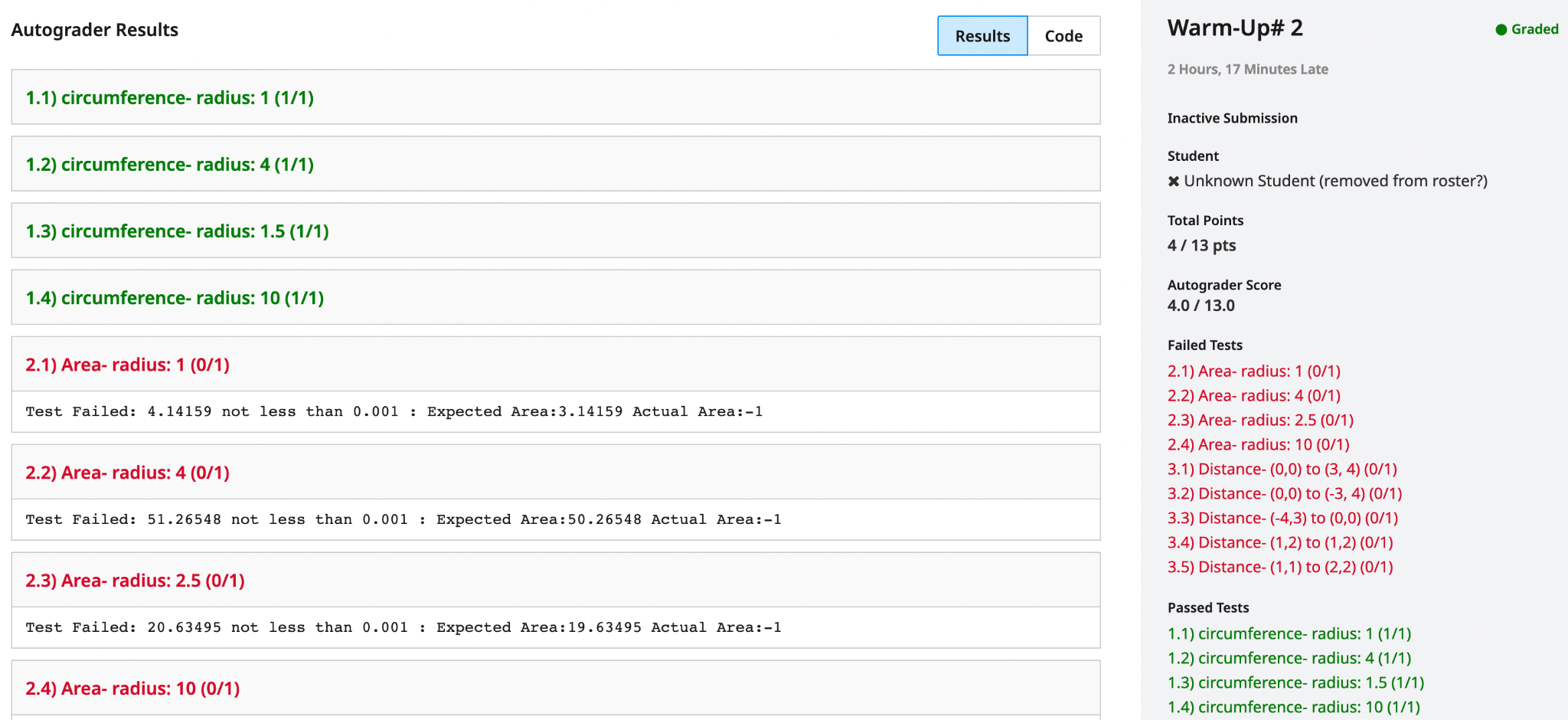
# **Python Warm-Up #2: Functions**

| **Submit the original** [**warm\_up\_2.py**](https://drive.google.com/file/d/1UGtMmJ89Lgtrr3NCCcNpK8Jn-mSy2Ni_/view?usp=drive_link) **to Gradescope.** |
| --- |

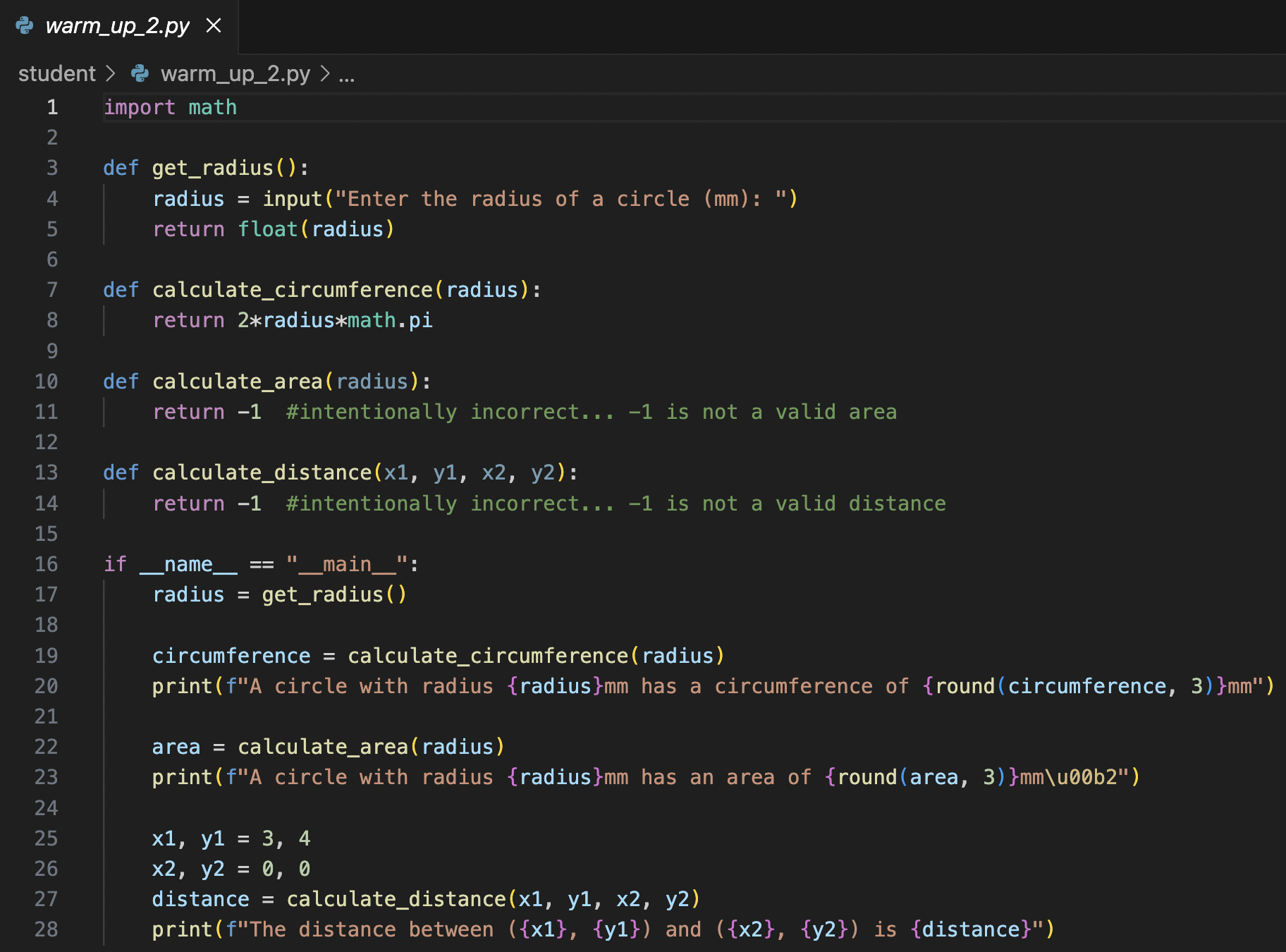
1. Visit the Gradescope Warm Up 2 assignment for your class.
2. Click on Submit assignment.
3. Drag the unmodified warm\_up\_2.py file to the submission box.
4. Click on Upload.
5. Notice how all 4 tests **Pass** for the first suite of test cases, while all tests **Fail** for the remaining two test suites.



Gradescope invokes your function implementations with a variety of input, comparing values that are actually returned with values we expect should be returned. This is known as **unit testing.**

| **2. Use the VS Code editor to read through warm\_up\_2.py.** |
| --- |

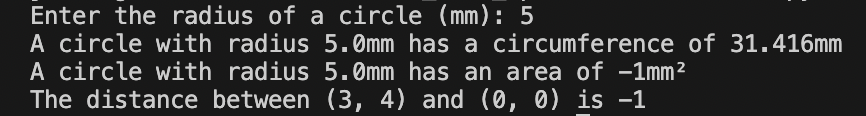
1. Read through the Python code in warm\_up\_2.py:



1. **What is the purpose of this program?**

| **3. Run warm\_up\_2.py with VS Code** |
| --- |

1. Run warm\_up\_2.py
2. Enter a radius of 5
3. You should see the following output in the VSCode Terminal window:



| **Understanding warm\_up\_2.py** |
| --- |

The "\_\_main\_\_" section of warm\_up\_2.py decomposes the process of finding the circumference of a circle into three smaller sub-problems:

1. Get a radius from the user.
2. Calculate the circumference of a circle using the given radius.
3. Display the resulting circumference to the user, with an appropriate context.

### 

### **Problem Solving with Code**

At a top level, many code-based problem solutions can be described by these three steps:

1. Gather necessary input.
2. Perform some calculation using the input.
3. Display the results.

warm\_up\_2.py implements the solution to two of these sub-problems in separate programmer-defined functions.

| **Sub-Problem #1: Get a radius from the user**radius = get\_radius() |
| --- |

Invoking the programmer-defined function **get\_radius** redirects program execution to the **get\_radius** function definition. Once **get\_radius** finishes, the returned radius data is stored in a variable called radius.

#### **Defining Python Functions**

In Python, function definitions must come before a function is used. Lines 3-5 provide the definition of the **get\_radius** function:

| **def** **get\_radius**():  radius = input("Enter the radius of a circle (mm): ")  **return** float(radius) |
| --- |

* The **header** in a function definition includes:
  + The keyword def is to begin the function definition
  + The name of the function.
    - Function names follow the same rules as variable naming.
  + ( )’s to ndicate any required or optional parameters
    - **get\_radius** has no parameters
    - ( )’s are also used to invoke (ie run) a function
  + The : symbol is to indicate the beginning of the body of the function definition.
* The **body** of a function definition includes:
  + The Python code which implements the function behavior
    - Consistent indentation indicates which lines of code should be considered part of the function definition
  + Optional return statement(s)
    - return immediately ends the function by sending data back to where **get\_radius**() was **invoked**.
    - None is the Python keyword used to indicate no value is returned from a function
    - Most of our functions will include return statements

#### **Python Data Types**

The radius and circumference of a circle are both numeric values. However, the Python input function only returns text-based, string data. This requires working with two data different types:

* float - Floating-point numbers (ie. decimals)
* str - Single-letters, and text-based words

Though there are other built-in Python data types, in this course we will focus on:

* Numeric Data Types: int, float
* Text Data Type: str
* Boolean Data Type: bool
* Sequence Data Types: list, tuple, range
* Mapping Data Type: dict
* Set Data Type: set

#### **Type Conversion**

***Type conversion***describes the process of converting data into a different data type.

The last line (Line 5) of **get\_radius**() converts string data returned by the input function into numeric data:

| **return** float(radius) |
| --- |

* This ensures that the data returned by the function is easily usable in arithmetic expressions

| **Sub-Problem #2: Calculate the circumference of a circle with the given radius** circumference = calculate\_circumference(radius) |
| --- |

A programmer-defined function called calculate\_circumference implements all the necessary details for this important step:

| **def** **calculate\_circumference**(radius):  **return** 2\*radius\*math.pi |
| --- |

* The body of the calculate\_circumference function uses both built-in arithmetic operators, and math library constants.

##### 

#### **Python Arithmetic Operators and the Order of Operations**

| Precedence | Operation | Operator | Example | Result |
| --- | --- | --- | --- | --- |
| 1 | **Override Precedence** | () | (2+1)\*4 | 12 |
| 2 | **Exponent** | \*\* | 2\*\*4 | 16 |
| 3 | **Multiplication** | \* | 2\*4 | 8 |
| 3 | **Division** | / | 2/4 | 0.5 |
| 3 | **Floor Division (Rounding down)** | // | -2//4 | -1 |
| 3 | **Remainder** | % | 2%4 | 2 |
| 4 | **Subtraction** | - | 2-4 | -2 |
| 4 | **Addition** | + | 2+4 | 6 |
| 5 | **Bitwise Shift Left** | << | 4<<2 | 16 |
| 5 | **Bitwise Shift Right** | >> | 4>>2 | 1 |
| 6 | **Bitwise AND** | & | 2&4 | 0 |
| 7 | **Bitwise XOR** | ^ | 6^4 | 2 |
| 8 | **Bitwise OR** | | | 2|4 | 6 |

#### **The math Library**

Line 1 in warm\_up\_2.py imports the math library: import math This allows the use math.pi as a reliably-accurate approximation for π.. The math library contains many useful functions/constants:

* constants: math.pi, math.e, etc.
* functions: math.sqrt, math.floor, math,ceil, math.log, etc.

math library functions and constants aren't available without first importing the math library.

#### **Functions and Unit Testing**

Even though the formula to calculate the circumference of a circle is essentially a 1-line calculation, creating a stand-alone function for calculate\_circumference is an important design decision which allows us to **unit test** the function.

**Unit testing** is the concept of testing code on a function-by-function basis with a robust collection of input..

Submitting warm\_up\_2.py to Gradescope runs three collections (called **suites**) of test cases, one for each programmer-defined function: calculate\_circumference, calculate\_area, calculate\_distance

This ensures that your function definitions anticipate a wide range of potential inputs:

1. Implementing the procedure for calculating a circumference as a function allows Gradescope to make several calls to calculate\_circumference using different values for the radius parameter.
2. Gradescope compares ***expected values*** with ***actual values***returned by the calculate\_circumference function to determine whether a particular test has failed.
3. The original copy of warm\_up\_2.py Fails two test suites because the two functions have not yet been implemented.

| **Sub-Problem #3: Display results to the user**print(f"A circle with radius {radius}mm has a circumference of {round(circumference, 3)}mm") |
| --- |

To help make results more readable, the built-in round function is used to indicate that only 3 decimals from circumference data should be displayed.

This output statement prioritizes **understandability** by providing a full context for understanding the calculated data. Important components of this include fully stating:

* the original problem
* important user input
* concisely displayed numeric results
* units for context

A poorly-considered version of this output would look something like:

| print(f"{circumference}") |
| --- |

This stand-alone numeric value will get lost in a sea of other information. Even for simple programs like warm\_up\_2.py it is relative easy to forget which numbers represent the circumference, area, or distance.

Making it a habit to prioritize understandability in your variable names and output statements can greatly improve the quality of your code.

| **Warm-Up #2 Exercises -** |
| --- |

### 

### **Exercise 2.1: Correct the implementation of** calculate\_area **in warm\_up\_2.py**

Save your changes, then submit **warm\_up\_2.py** to the Gradescope Warm Up 2 assignment on [Gradescope](https://www.gradescope.com/).

Resubmit via Gradescope until all tests pass for calculate\_area.

### **Exercise 2.2: Correct the implementation of** calculate\_distance **in warm\_up\_2.py**

Save your changes, then submit **warm\_up\_2.py** to the Gradescope Warm Up 2 assignment on [Gradescope](https://www.gradescope.com/).

Resubmit via Gradescope until all tests pass for calculate\_distance.