COMP151 - Function Designer's Checklist

Designing and developing functions is a multi-step process through which you incrementally develop your understanding of both the problem solved by the function and the inner workings of the function itself. This check list is adapted from the outstanding Design Recipe used in the text *How to Design Programs*[1] by Felleisen, et. al.

[1] see htdp.org

The Checklist

1. *Inventory the problem* and determine the function's ***signature***[2], and it's *purpose*[3]. Take note of functions that produce ***side-effects***[4]

   [2] The number and types of parameters and the type of return value(s)

   [3] What it's used for and how it's used

   [4] Action other than take and return values, i.e print, get user input, modify memory outside of the function, etc.

   Problem: Develop a function to compute the area of a right triangle from it's base and height lengths.

   This problem has two numerical parameters, the base and height of the triangle, and will compute another number, the area. The area very likely to be a `float`. Without any further clarification about base and height, it's probably safest to assume they too are `float` data. Given that nothing was said about printing or other effects, it seems this function is free of side-effects. All together we have:

   - *Signature:* Two floating point parameters (base and height), one floating point return value (area).
   - *Purpose:* To compute the area of a right triangle from its base and height.
   - *Side-Effects:* None.

2. Write the function definition's ***header***[5], a ***stub***[6] for the ***body***[7] that returns a value of the appropriate type, and document the signature and purpose with a proper ***docstring***[8].

   [5] The first line of the function def.

   [6] A placeholder statement. Typically just a return.

   [7] The indented statements after the header.

   [8] Multiline string that documents the purpose and signature of a function.

   The header follows pretty quickly from the signature. We already determined the types of parameters and the return value, so we mostly need to settle on the function and parameter names. The problem gives us good names for the parameters: base and height. The name of the function should reflect it's return value and purpose, we'll use *rt_area* for *right-triangle area* in case we have other area functions later. This leads to the following header: `def rt_area(base : float, height : float) -> float:`. The stub for the body can just be `return 0.0` as the function must return a float and that statement minimally satisfies that signature specification. Putting this all together and adding a docstring we get:

```
def rt_area(base : float, height : float) -> float:
  """
  Compute the area of a right triangle from it's base and height.

  Parameters:
    base : float - the length of the triangle base
    height : float - the length of the triangle height

  Return:
    float : the area of the triangle
  """
  return 0.0
```

3. Work out several specific ***use cases***[9] and for those use cases determine the function ***arguments***[10], expected return value(s), and expected side-effects. When prudent, document the use cases in the docstring as examples and code them up as runnable tests to aid what comes next.

[9] Concrete instance of using the function to meet its documented purpose.

[10] Actual values passed to the function

A quick check on the web, or the right math text, tells us that the area is $\frac{bh}{2}$ for base b and height h. A dead simple example would be a base and height of 1. This leads to an area of $\frac{1*1}{2} = \frac{1}{2}$ or 0.5. To compute this we'd call `rt_area(1,1)` and expect a return of 0.5. If we instead have a base of 1 and height of 2, then the call `rt_area(1,2)` should return $\frac{1*2}{2} = 1.0$. Finally a more interesting example might be a base of 3.5 and a height 2.25 where `rt_area(3.5,2.25)` will return $\frac{3.5*2.25}{2} = 3.9375$. If we're so inclined we might update the docstring to something like:

```
"""
Compute the area of a right triangle from it's base and height.
Examples:
    rt_area(1,1) -> 0.5
    rt_area(3.5, 2.25) --> 3.9375
Parameters:
  base : float - the length of the triangle base
  height : float - the length of the triangle height

Return:
  float : the area of the triangle
"""
```

These same examples could be written up as actionable test using print statement, e.g. `print(rt_area(1,2), 1.0)`, or a unit-testing framework.

4. *Implement* the function by replacing the stub body with code that actually meets the function's purpose. Use cases can and should be used to motivate generalized code form instance specific code, i.e. replace "hard coded" values with variable names. At this point

we should have some idea of what to do or at least where to start. If nothing else, working out the use-cases means we've solved the problem a few times and just need to translate what we did to code.

For all of our use-cases we multiplied base by height then divided by 2, all floating point operations. These leads to two reasonable versions of the body of our function:

```
# Option 1: compute area and store in variable, then return stored value.
area = base * height / 2
return area

# Option 2: drop the variable and do it in one line
return base * height / 2
```

Opting for option 2 we get an updated, and hopefully correct definition.

```
def rt_area(base : float, height : float) -> float:
  """
  Compute the area of a right triangle from it's base and height.
  Examples:
      rt_area(1,1) -> 0.5
      rt_area(3.5, 2.25) --> 3.9375
  Parameters:
    base : float - the length of the triangle base
    height : float - the length of the triangle height

  Return:
    float : the area of the triangle
  """
  return base * height / 2
```

5. Test completed function definition against use cases and other pertinent test-cases. Debug as needed. If, during testing, you discover that the signature is missing something then return to step 1 and update your understanding of the problem along with the header and docstring. We cannot know if our definition is correct until we actually use it. We should make every effort to test it outside of its use in a larger system. This could me writing unit-tests or loading the definition in a scratch file, calling the function on various arguments, and printing the result. Other options for testing exist, what's important is that you actually case the function to get called in python code and verify that it's behaving as expected on a variety of arguments.

   Run the following code and ensure the the printed return value is equal to the printed expected value (the second thing printed.)

```
print(rt_area(1,1), 0.5)
print(rt_area(1,2), 1.0)
print(rt_area(3.5,2.25), 3.9375)
```