UNIVERSITY OF WATERLOO

Lesson 1.4: Design Recipe in Python

Review: Design Recipe for Functions

Recall that when writing functions in Racket, we included the following design recipe elements:

- Purpose statement
- Contract
- Examples
- Function body
- Test cases

We will continue to include these elements for Python programs, but there will be some changes. The first of which being in Racket, the bulk of the documentation for a function occured before the function definition. While we could mimic this in Python, the more natural way to document code in Python is to include the information in a **docstring** after the function header.

Python's docstring

A **docstring** is a string that occurs immediately after the function header and contains information about the documentation for the current function. There is no universal standard for how this string should look and in this course, we will follow the conventions laid out in the previous course. Typically, the docstring begins with three single quotes and ends with three single quotes. This allows us to define strings on multiple lines with ease. It is possible however to have any string immediately following the header to be the docstring. For us, we will always use three single quotes.

The following is a simple example of a docstring with just a purpose statement.

```
1  def middle(a,b,c):
2      '''
3      Returns the median value of a, b, and c.
4      '''
5      largest = max(a,b,c)
6      smallest = min(a,b,c)
7      median = (a + b + c) - largest - smallest
8      return median
```

Visualize Reset Code

Updates to Our Design Recipe

The basic template for our design recipe remains the same however some of our verbiage will change to accommodate the standard jargon used in Python. Let's go through each element one by one to discuss the changes.

Purpose Statement

Our purpose statement still should clearly indicate what the function does, including explicitly mentioning the parameters and what they do. In Python, we use the word **returns** to describe what the output of a function is. As a result, we are retiring the use of **produces** in our purpose statements and will instead use the word **returns**. Notice also that we no longer need the function call at the beginning of the purpose statement because we have this information in our header!

Contract

As before, a contract describes the types of consumed (domain) and returned (codomain) values. Our contract should also include any additional requirements that are not already forced by specifying the domain of the function using our types. Most of the types are the same as in your previous course; however, we have deprecated types such as `Sym`, `Char`, and `Num`. As discussed in a previous lesson, we will use `Float` for non-integral numeric values.

Examples

Unfortunately, Python does not evaluate our examples as tests like Racket did. Thus, we will need to update our strategy in this course. Inside our docstrings, we will use the following for examples in our documentation:

```
fn(arg1, arg2, ...) => expected
```

As before, examples must include base cases as needed (this will become very important after we introduce `if` statements later in this course).

Body

The body is also a design recipe element. Most of what we will do in this course is develop new syntax to help us solve more complicated problems in Python. As this section will change throughout the course, we will leave this section brief now. One major change however is the use of `return` statements as was discussed at length in the previous lesson. The body of a function in Python should consist of statements which can include assignment statements, new local variables, calls to other functions - either built-in or user-defined, and other statements we will introduce in due time.

Comments can also be in the bodies of functions. To create comments in Python, use ## to write a comment on a full line or use # to write a comment at the end of a line. Comments should be used to help explain the **why** of your code. The **what** should be clear if you are using good design strategies like proper variable names and well labelled helper functions.

Statements should not exceed 80 characters from a style guide perspective however there is no actual enforced limit in Python. However, Python expects each line of code to be an entire statement. In order to allow for a user to write longer lines of code, Python lets users use the \ (backslash) character to delineate that a line will continue on the next line. This character is not needed if you have an open bracket on the previously unfinished line. Be sure to use this character as needed to extend your lines of code! This can even work for strings:

```
1  s = (" a very very very very very very very very very very"
2   " very very very very very very very very very very very very very very "
3   " very very very very very very very very very very very very very very long string!")
4
```

(Note you don't need the concatenation symbol + here) or you can use the backslash character:

```
1  s = " a very very very very very very very very very very" \
2 " very very very very very very very very very very very very very very very" \
```

```
" very very very very very very very very very very very very long string!"

```

While templates/code patterns, as were discussed in CS 115, will not be a focus in CS 116, you may still find them helpful, and we will try to point out common code patterns when it might be helpful.

Let's examine a few code samples with these four design recipe elements included. We still need to discuss how testing will work in this course which will be the bulk of the remaining discussion for this section. We include a few examples in the code so that you can run the examples in the visualizer.

Example 1

```
def middle(a,b,c):
    '''
    Returns the median value of a, b and c.

    middle: Int Int Int -> Int

    Examples:
       middle(4, 2, 8) => 4
       middle(3, 2, 1) => 2
    '''

    largest = max(a, b, c)
    smallest = min(a, b, c)
    median = (a + b + c) - largest - smallest
    return median

middle(4, 2, 8)
middle(3, 2, 1)
```

Visualize Reset Code

Example 2

```
def area(l, w):
    '''
    Returns the area of a non-trivial rectangular
    room of length l and width w.

    area: Float Float -> Float
    Requires:
       0.0 < l
       0.0 < w

    Examples:
       area(1.0, 1.0) => 1.0
       area(1.5, 8.0) => 12.0
    '''
```

```
15
16      return l * w
17
18  area(1.0, 1.0)
19  area(1.5, 8.0)
```

Visualize | Reset Code

Whitespace

Note the whitespace between the design recipe elements. In fact, your code should also include a header with your name and what assignment problem you are working on. Check the style guide for exact specifications on whitespace in your code and what a sample header should look like. When there is a discrepancy with what is correct in terms of style, the style guide will be taken as correct so you should periodically check the style guide to make sure you are keeping up to date with the standard and how it relates to new Python features we introduce.

As a reminder, our Style Guide can be found here.

Concept Check 1.4.1

1 point possible (graded)
Which of the following give reasons for why we use the Design Recipe? **Check all that apply.**

- [ ] It provides a place to start.
- [ ] Contracts and purpose can reduce simple syntax errors.
- [ ] Good design and template choices can reduce logical errors.
- [ ] Good design and template choices can provide better solutions.

Testing

In Python, there is, once again, no universal standard for testing. For our purposes, the existing software to test is not sufficient for our applications. Hence, we have created our own testing module for this course called the `check` module.

> A module is a collection of functions and, in this case, is a collection of functions we will use for testing.

The `check` module will be included in every single problem you solve on this website which makes things slightly easier. After each test, a message is displayed that confirms the test has passed or that it has failed. All of our tests will go after the body of the function and should be the last lines of code in any file you are submitting for assignments.

If you are working offline or working on Python code after this course, there is a copy of our `check` module that we make available here. You can download this file and copy it to the **same folder** of the Python file you are currently working on. To use this module offline, you will also need to include the line `import check` at the top of your file (more on this in the next lesson).

Just like in Racket, we will have two different tests: `check.expect` and `check.within`.

`check.expect`

Our `check.expect` function has 3 parameters.

1. The first parameter is the name of the test. This should be a descriptive name describing what the test should be testing.
2. The second parameter should be the function call. This is the actual output of your function on your test.
3. The third parameter is the actual output of the function. This should be computed by hand. Never run a function first to determine this output value!

The `check.expect` function should be used whenever we have exact data types **in our output** such as `Nat`, `Int`, or `Str` types. We will add other exact types later in the course.

Concept Check 1.4.2

1 point possible (graded)
Which of the following correctly uses the `check.expect` function with our `middle` function above?

- `check.expect("Test neg", middle(-1, 2, 3, 2))`
- `check.expect("Test neg", middle(-1, 2, 3), 2)`
- `check.expect(2, middle(-1, 2, 3), "Test neg")`
- `check.expect("Test neg", 2, middle(-1, 2, 3))`

More on Testing

`check.within`

Recall that not all data types in Python are exact. In fact, when testing with `Float` values, you need to use a different function for testing. This is because `Float` values are not stored exactly and so rounding errors can occur in computations. Thus, we don't want to check for exactness, but rather to check for correctness **within some tolerance**. The default tolerance in assignments is `0.00001`.

To do these tests, we use the `check.within` function in our `check` module. This takes in the same three parameters as `check.expect` but also includes a fourth parameter for the tolerance.

It is important to stress this:

Never use `check.expect` with `Float` values!

Even if you think the answer should be exact, there is always the possibility of inexactness when using `Float` values. As another reminder, do not use `Float` values unless you absolutely have to and try to use arithmetic operations over the integers to avoid the inexactness of computations involving floats.

The function `check.within(name, expr, value_expected, tolerance)` performs the test:

```
abs(expr - value_expected) <= tolerance
```

As it is not a precise check, it should only be used for inexact **output** values from functions (like `Float`) and never for exact output values (like `Int` or `Str`).

Examples As Tests

Note: Your examples should be coded as tests as well. This point is important enough to have its own short heading. You will have to translate your examples from the docstring to the end of your code and rewrite them using the `check` module.

With all the above, we can write down two fully complete examples using our full design recipe. Please note that the style guide has many other examples and should be the go to reference if you are confused about how to use the design recipe in this course.

Example 1 Complete

```
import check


def middle(a,b,c):
    '''
    Returns the median value of a, b, and c.

    middle: Int Int Int -> Int

    Examples:
       middle(4, 2, 8) => 4
       middle(3, 2, 1) => 2
    '''

    largest = max(a, b, c)
    smallest = min(a, b, c)
    median = (a + b + c) - largest - smallest
    return median

##Examples:
check.expect("Example 1", middle(4, 2, 8), 4)
check.expect("Example 2", middle(3, 2, 1), 2)

##Tests:
check.expect("Test all equal", middle(0, 0, 0), 0)
check.expect("Middle at end", middle(3, 1, 2), 2)
check.expect("All negative", middle(-3, -1, -2), -2)
```

Visualize Reset Code

Example 2 Complete

```
import check

```

```
def area(l, w):
    '''
    Returns the area of a non-trivial rectangular
    room of length l and width w.

    area: Float Float -> Float
    Requires:
       0.0 < l
       0.0 < w

    Examples:
       area(1.0, 1.0) => 1.0
       area(1.5, 8.0) => 12.0
    '''

    return l * w

##Examples:
check.within("Example 1", area(1.0, 1.0), 1.0, 0.00001)
check.within("Example 2", area(1.5, 8.0), 12.0, 0.00001)

##Tests
check.within("Tiny area", area(0.000001, 0.000001), 0.0, 0.00001)
check.within("Large area", area(100000.0, 100000.0), 10000000000.0, 0.00001)
check.within("Fractional area", area(1.5, 2.25), 3.375, 0.00001)
```

Visualize Reset Code

Concept Check 1.4.3

1 point possible (graded)
Which of the following return types for functions should be used with `check.expect`? **Check all that apply.**

- [ ] `Nat`
- [ ] `Int`
- [ ] `Float`
- [ ] `Str`

Concept Check 1.4.4

1 point possible (graded)
Consider the following code:

```
1  import check
2
3  a = 10**116
4  b = int(10**116/1)
5  check.expect("Testing Arithmetic", a, b)
```

True or False: The above test passes

- ◯ True
- ◯ False

Concept Check 1.4.5

1 point possible (graded)
Consider the following code:

```
1  import check
2
3  def add_one(x):
4    return x + 1
5    check.expect("Testing 0 + 1", add_one(0), 1)
```

True or False: The above test is written correctly

- ◯ True
- ◯ False

Even More on Testing

We have explained how to test functions in this course, but what is an appropriate set of tests (called a *test suite*) for a function? There is not a precise answer, but there are some underlying principles to keep in mind.

For most functions, there are an infinite number of possible inputs. No quantity of tests will cover all possibilities. Instead, we focus on the **quality** of a test suite. When selecting test data for a function, take the attitude that the function is incorrect and your job is to find at least one case that demonstrates this. To use a courtroom analogy, treat the function as the defendant and the tester's role as the prosecutor. This perspective is especially important when testing code you wrote yourself.

Black-box tests

There is a downside to testing your own code. It can be tempting to assume it is correct and errors may be unlikely to be revealed if tests are based on your particular approach to a problem. Alas, *black-box* tests are designed based only on the purpose and contract of a function. This is one of many reasons why it is a good idea to complete the body of a function **after** other elements of the design recipe.

When creating black-box tests, consider all the parameters and the return value. Ensure your test suite includes:

- typical values (e.g. the test of the second example of `area` in Example 2)
- small and large values (e.g. return values of `area` representing a tiny area and a large area)
- boundary values (e.g. if `middle` in Example 1 only consumed natural numbers, a case where `a` is zero, a case where `b` is zero and a case where `c` is zero)
- special cases (e.g. for `middle`, when the middle value is each of the first second and third values)
- how the parameters relate to each other (e.g. for `middle`, we test when the three values are all equal)

Ideally, these situations should be tested both independently and in different combinations, but when evaluating tests on assignments or concept checks in CS 116, we will only check that you cover an appropriate set of situations.

White-box tests

Some tests can and should be designed based on inspection of the code being tested. This is called *white-box testing*. At a minimum, this involves ensuring that test data causes all lines of code to execute. As we learn about different control structures in Python, we will see that code is not always executed in a simple linear fashion. It will then become important to consider different possible paths through the lines of code. Naturally, white-box tests can only be written after the body of a function has been completed.

Concept Check 1.4.6

The function `in_interval(start, end, val)` consumes two natural numbers `start` and `end` representing a closed interval on the non-negative x-axis, and a natural number `val`. The function determines whether or not `val` is in the closed interval. That is, 1 is returned if `val` is between the endpoints of the interval or at the endpoints of the interval. Otherwise, 0 is returned.

(In Module 2, we will introduce a new type which is a more appropriate return type for this situation.)

```
1  def in_interval(start, end, val):
2      '''
3      Returns 1 if val is in the interval [start,end].
4      Otherwise returns 0.
5
6      in_interval: Nat Nat Nat -> (anyof 0 1)
7
8      Requires:
9          start <= end
10
11     Examples:
12         in_interval(1,5,3) => 1
13         in_interval(1,5,7) => 0
14     '''
15     ##BODY OF FUNCTION IS HIDDEN
```

We have started a black-box testing suite for `in_interval` using the `check` module. It is incomplete because it misses three distinct situations outlined above. Add distinct tests to complete the test suite so that it adequately covers the cases that should be tested for this function.

All of your tests must agree with the contract and requirements of `in_interval`.

Note that we are choosing meaningless labels for our tests to avoid giving anything away.

0/3 points
Attempts: 0 / Unlimited

```
#the check module is automatically imported for this question
check.expect("Example 1", in_interval(1,5,3), 1)
check.expect("Example 2", in_interval(1,5,7), 0)
check.expect("Test 1", in_interval(2,9,2), 1)
check.expect("Test 2", in_interval(2,9,9), 1)
check.expect("Test 3", in_interval(0,8,4), 1)
check.expect("Test 4", in_interval(0,7,0), 1)
check.expect("Test 5", in_interval(10**6,10**8,10**7), 1)
check.expect("Test 6", in_interval(10**6,10**8,10**10), 0)
```

Helper Functions and Design Recipe

As was the case with Racket, helper functions are strongly encouraged in this course. All helper functions (unless a question explicitly dictates otherwise) should have a contract, purpose, and body. Examples and tests are not required for helper functions in this course, though both are encouraged and recommended to help guide your answer and to check for correctness. Helper functions can be defined locally, though we will discuss this later and, in general, we discourage this use of local helper functions in this course. Regardless of how helper functions are defined you still need purpose, contract, and the body of the function to be included.

Final Thoughts

In this lesson, we discussed all of the updates our design recipe needed in order to be compatible with Python. Each section had changes including:

- Purpose statements use the word **returns**.
- Contracts don't include `Num` types.
- Examples need to be rewritten and duplicated as tests.
- Our body will use a new syntax entirely and make use of the `return` statement.
- Testing uses our homebrew `check` module.

In the last lesson, we introduce another module called the `math` module. This will need to be included on every problem (unlike the `check` module, which we include for you in every problem in this course).