✓ Done

Tutorial 5

Debugging and Testing

Introduction

The purpose of this practical is to continue using Python to develop programs, concentrating on how to detect, fix, and prevent errors. Don't forget that you are required to submit the indicated exercise through myLearn for assessment.

Errors

Everybody makes mistakes when they code. It might just be a simple typo, or it might be a more concerning logical error. In the best case, the compiler/interpreter will notice the error and notify the programmer. Unfortunately, those error messages aren't always very clear. Other times, the program might compile and run just fine - only not do what we expect of them. This week we examined some of the error types in Python programs, and methods to detect and avoid them. We'll use some of those techniques in this week's exercises.

Exercises

Use material from this week's lectures to perform the following exercises:

1. **[This exercise should be submitted through the Assessments section of myLearn for Tutorial Exercise 5]** Debug the following program:

```
Def factorial(n):
  """Calculate the factorial of the given value and return the result.

The factorial of n is the product of all positive integers less than or equal to n.
This function does not support negative values - if a negative value is given, this function just
returns 1.

Arguments:
n -- A positive integer
  """
  result = 1
  while n != 0:
    n = n - 1
    result = result * n
  return result

# Calculate factorial for the first four integers
for i in range(-1, 5):
  print('Factorial of', i, 'is', factorial(i)
```

2. Add assertions to your answer to 1) that ensure the assumptions made in the code are enforced

3. Write unit tests for the program specified below. You should have a separate test function for each possible behaviour.

    1. Assume the program will be written in a file called *X.py* (so your unit test file should begin with `import X`)
    2. The program will have the following functions:

        - `add(x, y)`: adds `y` to `x` and returns the result
        - `sub(x, y)`: subtracts `y` from `x` and returns the result
        - `mul(x, y)`: multiplies `x` by `y` and returns the result
        - `div(x, y)`: divides `x` by `y` and returns the result

- `calculate_strange_value(x, y)`: If `y` is greater than `x`, returns the productof `mul(x, y)` and `add(x, y)`. If `x` is greater than `y`, returns the product of `div(x, y)` and `sub(x, y)`. If `x` and `y` are equal, returns `1`
- `is_valid_positive_integer_input(value)`: Returns `True` if the string `value` can be correctly converted to a positive integer, `False` otherwise

3. When run, the program will ask the user to enter two positive integers (and will repeatedly ask until they have done so). It will then display the result of `calculate_strange_value` with those two integers as arguments and exit

4. Use TDD to write the program specified by your unit tests in 3). To do so, find your first unit test that fails, then write code to make that test pass. Once the first test is passing, add code to make the second test pass too (i.e. after making changes, test the code is passing the first two tests before continuing). Continue this process (incrementally getting one more test to pass) until all tests pass. Include assertions for the parameters in each of the functions to make sure they are the right type.

Last modified: Thursday, 1 February 2024, 10:51 AM