Computer Science 1 — CSCI-141
Testing and Debugging — Homework 6

8/26/2020

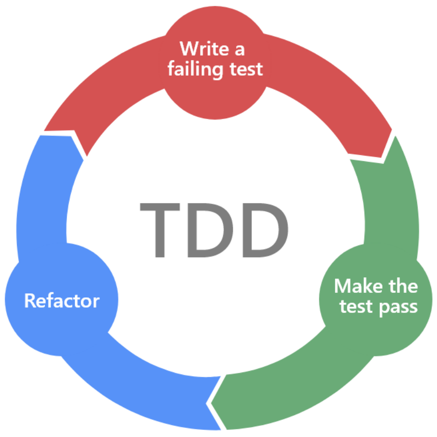

Figure 1: Test-driven development cycle

1 Overview

In lecture this week you learned about:

- a software development approach, *Test-driven development*, where tests are developed initially, and then the production code is developed incrementally and possibly refactored.
- how to use the *debugger* in PyCharm - a tool that allows you to test and debug your *target* program/s.

Please note, this entire assignment must be done using PyCharm. If you haven't set it up yet, please follow these instructions:

`https://www.cs.rit.edu/~csci141/Docs/PyCharm-Setup.html`

2 Task 1: Ackermann Function

The Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive. Don't worry about what those terms mean it is not important for this assignment.

The most common version is a two-argument version defined as the following for non-negative integers, `m` and `n`:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

2.1 Starter Code

Create a new project for this homework assignment and add the following test module, `test_ackermann.py` to it:

`https://www.cs.rit.edu/~csci141/Homeworks/06/test_ackermann.py`

2.2 Function Stubbing

The first step in development is to create a new Python source file, `ackermann.py`. This module is referred to as *production code* and is intended for end users. To begin with, you will not implement the `main` function.

The `ack` function takes two non-negative integers, `m` and `n`, as its parameters. It returns an integer result. To begin with, implement this function in `ackermann.py` and have it return a value of `0`.

Now run the test module `test_ackermann.py`. You should see all 15 tests compile and run, but they all fail.

```
ack(0, 0) failed; expected 1 but got 0
ack(0, 1) failed; expected 2 but got 0
ack(1, 0) failed; expected 2 but got 0
ack(1, 1) failed; expected 3 but got 0
ack(1, 2) failed; expected 4 but got 0
ack(1, 3) failed; expected 5 but got 0
ack(2, 1) failed; expected 5 but got 0
ack(2, 2) failed; expected 7 but got 0
ack(2, 3) failed; expected 9 but got 0
ack(3, 1) failed; expected 13 but got 0
ack(3, 2) failed; expected 29 but got 0
ack(3, 3) failed; expected 61 but got 0
ack(4, 0) failed; expected 13 but got 0
ack(3, 4) failed; expected 125 but got 0
```

2.3 Fail / Pass / Refactor

1. Implement **only the base case (when `m = 0`)** from the definition of the Ackermann function into your `ack` function in `ackermann.py`. Leave the `return 0` statement at the end since all the cases are not implemented yet. Re-run your `test_ackermann.py` program and you should see that it now passes the first two test cases.

2. Implement **the first recursive case (when `m > 0` and `n = 0`)** from the definition of the Ackermann function into your `ack` function in `ackermann.py`. Again leave the `return 0` statement at the end. Re-run your `test_ackermann.py` program and you should see that it now passes the first three test cases.

3. Implement **the second/final recursive case (when `m > 0` and `n > 0`)** from the definition of the Ackermann function into your `ack` function in `ackermann.py`. You can now remove the `return 0` statement at the end since all the cases are implemented now. Re-run your `test_ackermann.py` program and you should see that it now passes all test cases.

2.4 Production Main

Now implement a `main` function in the `ackermann` module. It should be protected with an *if main guard* at the bottom of your file (cutting and pasting this may cause issues), i.e.:

```
def main():
    # main functionality goes here

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

1. The function should first prompt the user to enter two integers, `m` and `n`. You may assume the user enters valid arguments for the `ack` function that will cause it to run in a reasonable amount of time.
2. Call your `ack function` with the arguments and get the result.
3. Display the result.

Run the main program to create a new run configuration for `ackermann` and make sure it behaves as described above.

2.4.1 Sample Run

Here is a sample run of the `ackermann` module's `main` function.

```
Enter m: 3
Enter n: 2
ack(3, 2) = 29
```

3 Screen Shots

The next task will require you to take screenshots of your PyCharm debugging window that you need to save and submit along with your code changes. Follow these directions for your operating system to make sure you are able to take and save a screen shot.

To begin with, make sure your PyCharm project is open.

3.0.1 Windows 10

1. In the search bar to the right of the start menu, type **Snipping Tool**.
2. The Snipping Tool application should show up as the best match - click it to launch.
3. In the **Mode** pull-down menu, select **Window Snip**.
4. Click **New** to open the camera and move your mouse over your PyCharm window.
5. Left click in the window and the screen will be captured.
6. Go to the **File** pull-down menu and select **Save As**.
7. Select the location on your computer the image will be saved (by default it is **This PC/Pictures**).
8. Click on **File name** and name your file **test.png**.
9. Click the **Save** button to save the image.

10. Navigate in a file explorer to where you saved the image, open it in Photos and verify it.

3.0.2 Mac OS/X

1. Press these four keys together, **Command** + **Shift** + **4** + **Space**.
2. The window cursor should change to a camera icon. Move it to your PyCharm window and click your left mouse button.
3. The image will save on your desktop as **Screen Shot...png**.
4. Click on the text of the filename on your desktop and rename the file to **test.png**.
5. Double click on the file to open it in Preview and verify it.

4 Task 2: Palindrome Checker

A *palindrome* is a word, number or phrase that reads the same backwards and forwards, e.g *racecar*, *madam*, *amanaplanacanalpanama*.

4.1 Starter Code

Add the following module to your existing PyCharm project (in the same location as your other source code):

`https://www.cs.rit.edu/~csci141/Homeworks/06/palindrome.py`

This module is *production code* for a palindrome checker that is intended for an end user. Some users have reported to you that sometimes the program does not work as intended and incorrectly identifies some words as palindromes when they are not. To assist you with finding the source of the problem and eventually fixing it, you will employ a testing methodology.

4.2 Test Module

In order to find and then fix the problem/s that exist, you will implement a new test module, `test_palindrome.py`, with a series of well defined test cases for the `is_palindrome` function in `palindrome.py`.

4.2.1 Step 1: Stubbing

1. Create a new Python source file, `test_palindrome.py`, in the same place as all your other source code.
2. Using `test_ackermann.py` as a reference, set up your test module initially with the following:
   (a) An *if main guard* that calls the `run_tests` function.
   (b) The `run_tests` functions has no arguments and returns nothing. Initially it can be implemented with a single statement, `pass`.

(c) Add a `test_is_palindrome` function that will be used to test a single call to the `is_palindrome` function in the `palindrome` module. This function should take three arguments, a string with the function and arguments being tested, the string to be tested for being a palindrome or not, and the expected result of the test (True if it is a palindrome, False otherwise). Initially it can be implemented with a single statement, `pass`.

(d) At the top of the module add a statement to import the `is_palindrome` function from the `palindrome` module, i.e. `import palindrome`. To call the function you need to say `palindrome.is_palindrome(...)`.

3. Run your `test_palindrome` module to create a run configuration. It should compile and run without warning and produce no output.

4.2.2 Step 2: Palindrome Tester

1. Fully implement the `test_is_palindrome` function in the `test_palindrome` module. It will look somewhat similar to the `test_ack` function in `test_palindrome.py`. It needs to call the `is_palindrome` function in `palindrome.py` with the `string` to be tested, and print whether the `result` matches the `expected`.
2. Now in `run_tests`, you should create **at least eight test cases** with strings that are palindromes and are not palindromes. **At least one of your tests should be a string that is six characters in length that is not a palindrome!** Use the `run_tests` function from `test_ackermann.py` for reference. **Hint: Make sure you test with strings of many different lengths! Also, do NOT use real words; just use combinations of the characters `a` and `b`!**
3. Re-run your `test_palindrome` module. **You should have a minimum of two test cases that fail.** If all your tests pass, keep modifying your test cases until two fail (one must be a string of six characters that is not a palindrome, but tests as if it were).

4.2.3 Step 3: Using the Debugger

Now that you have test cases that fail, it is time to use the PyCharm debugger to identify where the problem is.

1. Set a breakpoint in your `run_tests` function of the `test_palindrome.py` on the line of code with the test call for the string of six characters that fails. Left click in the column with the line number and add a red breakpoint circle should appear.
2. Run your `test_palindrome` module in debug mode by selecting the green bug icon directly to the right of the green run icon by the run configuration.
3. The debugging window should come up below your code. Take a screen shot of your entire PyCharm window and save the file as `first.png`.
4. Next, go to the `is_palindrome` function in `palindrome.py`. Set a breakpoint on the line with the first `if` statement.
5. Resume debugging by clicking on the green **Resume Program** icon in the debugger controls on the far bottom left. The debugger will now step into the first call to the `is_palindrome` function. Take a screen shot of your entire PyCharm window and save the file as `second.png`.

6. We know this test is returning `True` for a string that is not a palindrome. Set a third breakpoint on the `return True` statement and repeatedly resume the debugger until the statement is encountered. Take a screen shot of your entire PyCharm window and save the file as `third.png`. Make note of the value of the string here.

4.2.4 Step 4: Fixing the Code

Using the information from your third screenshot, i.e. the stack trace and the values of the variables, try and fix the `is_palindrome` function in `palindrome.py` so it behaves correctly. Keep re-running your test module, modifying `is_palindrome`, and debugging until the test module successfully passes all the tests.

5 Grading

- Task 1: Ackermann Function, `ackermann.py`
  - `ack` function: 15%
  - `main` function: 15%
- Task 2: Palindrome Debugging
  - `test_palindrome.py`:
    - `run_tests` function: 20%
    - `test_palindrome` function: 20%
  - `palindrome.py`:
    - `palindrome` function: 10%
  - Screen shots: 10%
- Function docstrings (Task 1 and 2): 5%
- Module docstring (`ackermann.py` and `test_palindrome.py`): 5%

6 Submission

1. Create a folder on your desktop named `hw06`.
2. Navigate in an explorer to your PyCharm project and copy (do not move!) the following files into your `hw06` folder:
   - `ackermann.py`
   - `palindrome.py`
   - `test_palindrome.py`
3. Navigate in an file explorer / finder to where you saved your three screen shots from the debugging session and copy them into your `hw06` folder:
   - `first.png`
   - `second.png`
   - `third.png`
4. Zip your `hw06` folder into a file named `hw06.zip`.
5. Upload your zip file to the Assignment dropbox by the due date.