CSC 110
CSC110 Intro to CS through programming
Homework 01

Spring 2025

Variables and Operations

`Due: By next Friday, BEFORE CLASS`

Note: Homework Assignment 1 should be completed individually.

Instructions

Read the instructions for parts 1 through 4. Note that at the end of the instructions there is a section reminding you where stuff is and how to run the tests.

Then, you can start working on the Homework.

Notes :

- All your editing will happen in the file called hw01.py.
- As you advance, make small edits and then run the program, in a Bash shell, with instruction:

```
python3 hw01.py
```

- Deal with any issues (syntax errors, runtime errors, or logical errors) and then continue.
- Once the program compiles and you have succeeded in completing one part, run the tests to verify you are now passing some of them:

```
python3 hw01.py
```

Part 1

Add code to the part 1 section after the line:

```
# Your code for part 1 under this line and before the print statements
```

and before the comment:

```
# End of Part 1 ----------------------
```

Overwrite the required variables so that you complete the following steps (in python):

- assign 27 to a variable named x
- assign 1 to a variable named y
- assign 1.5 to a variable named a
- assign 7 to a variable named b
- assign -1 to a variable named c

Note that you do not modify the initial assignments at the top, where all variables are initialized with zeros. Instead you need to add new assignments where you overwrite their values with the ones indicated above.

Then, you need to write an arithmetic expression, in python so you obtain the following:

$result1 = \frac{3x-9y}{2a(b-c)}$

At the end, you need to add print statements so that the following is printed:

```
Part 1: x = 27
Part 1: y = 1
Part 1: a = 1.5
Part 1: b = 7
Part 1: c = -1
Part 1: result = 3.0
```

Do not just assign the value of 3.0 directly to result1, make sure the expression results in 3.0

There are tests in the test section to check if you did this correctly (more on tests at the end of these instructions)

Part 2

Add code to the part 2 section after the line:

```
# Your code for part 2 under this line and before the print statements
```

and before the comment:

```
# End of Part 2 ----------------------
```

Overwrite the required variables so that you complete the following steps (in python):

- assign 5 to a variable named x
- assign -3 to a variable named y

Note that you do not modify the initial assignments at the top, where all variables are initialized with zeros. Instead you need to overwrite their values with the ones indicated above.

Then, you need to write an arithmetic expression, in python so you obtain the following:

$result2 = x^2y^4$

At the end, the print statement should print:

At the end, you need to add print statements so that the following is printed (after the previous part's printouts):

```
Part 2: x = 5
Part 2: y = -3
Part 2: result = 2025
```

Do not just assign the value of 2025 directly to result2!

There are tests in the test section to check if you did this correctly (more on tests at the end of these instructions)

Part 3

We need to solve a problem of "fair distribution". Let's say you have 100 treats and 13 dogs. Now, we want to give an equal number of COMPLETE treats to each dog. Here are the steps to perform the calculations.

Add code to the part 3 section after the line:

```
# Your code for part 3 under this line and before the print statements
```

and before the comment:

```
# End of Part 3 ----------------------
```

Overwrite the required variables so that there is one line of code for each of the following steps:

- assign 100 to a variable named a (representing the total number of treats)
- assign 13 to a variable named b (the total number of dogs)

Then, you need to write an arithmetic expression, in python so you obtain the following:

$result3$ = the number of whole (integer) treats that each dog gets

At the end, you need to add print statements so that the following is printed (after the previous part's printouts):

```
Part 3: a = 100
Part 3: b = 13
Part 3: result = 7
```

Do not just assign the value of 7 directly to result3!

There are tests in the test section to check if you did this correctly (more on tests at the end of these instructions)

Part 4

We need to solve a problem of calculating leftovers. For the problem solved above (100 treats split evenly into 13 dogs, without breaking up treats), there will be a leftover. We need to calculate this amount using Python.

Add code to the part 4 section after the line:

```
# Your code for part 4 under this line and before the print statements
```

and before the comment:

```
# End of Part 4 ----------------------
```

Overwrite the required variables so that you complete the following steps (in python):

$result4$ = the number of whole (integer) leftover treats that we have after giving each dog an equal number of treats.

Note : There is no need to reassign a and b, we'll use the previous values (from part 3) Hint : Use Modulo!

At the end, you need to add print statements so that the following is printed (after the previous part's printouts):

```
Part 4: result = 9
```

Do not just assign the value of 9 directly to result4!

There are tests in the test section to check if you did this correctly (more on tests at the end of these instructions)

Adding one instruction to the program

The Python program has a a header section you should fill in (replacing the placeholder info). The code has comments indicating where to add your code. Remember, "comments"... are text that do not get executed.

Expected Output

You execute your program in Thonny or on the terminal. In the terminal, you can run it with: `python3.13 hw01.py`

This is what the output should look like when you execute your program:

```
Part 1: x = 27
Part 1: y = 1
Part 1: a = 1.5
Part 1: b = 7
Part 1: c = -1
Part 1: result = 3.0
Part 2: x = 5
Part 2: y = -3
Part 2: result = 2025
Part 3: a = 100
Part 3: b = 13
Part 3: result = 7
Part 4: result = 9
```

Tests and checking your work

In addition to the hw01.py file, where you will make edits to complete the homework, you might see additional python files called test_<something>.py. You are not to modify these files (you don't even need to look at it). These files are provided so you can see how many tests you have passed so far. If you pass all tests, then your code is complete.

To run the tests, you run a special python module called pytest. It is explained in the section: **Testing**.

Grading criteria:

The submission:

- runs without syntax errors (or -50%)
- adds a few small but informative comments (or -10%)

The program:

- Passes all 13 tests (or lose 5% per missed test).

Submitting

The way you submit is to make sure you've pushed all changes to your remote repository. I will collect all of your hw01 repositiories at the due date/time and can look into them and test them directly.

Testing

The word "test" when talking about programs, normally refers to the act of checking if it works correctly. The way this is done is to execute a tester program that calls your program and that checks for expected behavior.

Note that, you only perform "testing" once the program actually compiles.

So, the process should be:

1. add code that performs one small step.
2. execute the file and verify the program compiles and does not crash.
    1. If it doesn't compile or if it crashes, fix the syntax errors and logic errors until it doesn't have any errors and the base program compiles (even if it is still incomplete)
3. Repeat until you have completed a "feature". Normally, a feature is one distinct step ot task in the full program. For this homework, it's each of the parts.
4. Once a feature is completed, test the program. You will see FAILURE for any tests that have to do with incomplete features, but you should see any feedback (and hints) about features you already worked on that fail. If they fail, check the logic and correct any differences with the expected results. If they pass, move on to the next feature.

For this class, you don't need to create or modify the test programs, but you should use them to check if the expected result is achieved.

How to test

To test, follow these steps:

1. open a terminal at the location of the file you want to test. 1. In addition to the testable file (hw01.py, in this case), there should be a tester file (test_hw01.py).
2. run the following command: `python3.13 -m pytest -v -s`
3. Interpret the output to make changes in your hw01.py file.

How to Interpret the Test printout

Check the slides at the end of Lecture04. It has a visual explanation.

the way the tests are run is by executing the command: `python3.13 -m pytest -v -s` in your terminal. What this is saying is: Use python3.13 to run the pytest module (`-m pytest`) with verbose output (`-v`) and display output to the terminal (`-s`) If you don't have it installed, reach out to your instructor.

Pytest executes a test file (`test_hw01.py`) that accompanies your homework python file (`hw01.py`) and it displays the results to the terminal window (It's best to make the window wide to read them easily).

Imagine you've added the following code in the Part 1 section:

```
# Part 1: Basic Operations
# ========================================
# Your code for part 1 under this line and before the print statements
x=27
y=1
a=1.5
b=7
c=1
result1 = (3 - 9*y)/(2*a)*(b-c)
print("part 1: x =",x)
print("Part 1: y =",y)
print("Part 1: a = ",a)
print("Part 1: b =",b)
print("Part 1: c =",c)
print("Part 1: result =", result1)
# End of Part 1 ----------------------
```

This has many issues:

1. the value for c should be negative
2. the expression for result1 has many errors
3. the printout for x is NOT what is expected
4. the printout for a is NOT what is expected
5. the printout for c is NOT what is expected
6. the printout for result1 is NOT what is expected

When you run the command: `python3.13 -m pytest -v -s`

Pytest printout:

This is the TOP part of what you'd see. Note: For this explanation, ignore the terminal printout that goes beyond the right margin.

```
pfrank@Pablos-MBP hw01 % python3.13 -m pytest -v -s
============================= test session starts ==============================
platform darwin -- Python 3.13.1, pytest-8.4.1, pluggy-1.6.0 -- /usr/local/bin/python3.13
cachedir: .pytest_cache
rootdir: /Users/pfrank/Library/CloudStorage/Dropbox/Pablo/Smith/Academic/Teaching/CSC110/csc110-25f/homewo
collected 13 items

test_hw01.py::test1_x_printout FAILED
test_hw01.py::test1_y_printout PASSED
test_hw01.py::test1_a_printout FAILED
test_hw01.py::test1_b_printout PASSED
test_hw01.py::test1_c_printout PASSED
test_hw01.py::test1_result1_printout FAILED
test_hw01.py::test2_x_printout FAILED
test_hw01.py::test2_y_printout FAILED
test_hw01.py::test2_result2_printout FAILED
test_hw01.py::test3_a_printout FAILED
test_hw01.py::test3_b_printout FAILED
test_hw01.py::test3_result3_printout FAILED
test_hw01.py::test4_result4_printout FAILED

================================== FAILURES ==================================
```

To see how the test system helps, its a good idea to read on and look at the output for each specific test.

Here is the output for the section related to the test called `test1_x_printout`

```
_______________________________ test1_x_printout ________________________________

capsys = <_pytest.capture.CaptureFixture object at 0x10966da90>

    def test1_x_printout(capsys):
        hw01.main()
        captured = capsys.readouterr()
>       assert "Part 1: x = 27" in captured.out, "Tip: check the printout for x is EXACT"
E       AssertionError: Tip: check the printout for x is EXACT
E       assert 'Part 1: x = 27' in 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPart
E        +  where 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPart 1: c = -1\nPart

test_hw01.py:8: AssertionError
```

It looks like a lot, but don't worry, it's less crazy than you think. This tells you which part of the test code (that you did not write) detects the error. The issue is detected when running the `assert` statement, which just checks that something is happening. In this case, the expected outcome does not happen and that's why it complains.

The complaint is shown here:

```
E       AssertionError: Tip: check the printout for x is EXACT
E       assert 'Part 1: x = 27' in 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPart
E        +  where 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPart 1: c = -1\nPart
```

If you read carefully, it prints a hint: "Tip: check the printout for x is EXACT", and it also states where it fails. This part looks like a lot, but what the assert is doing is checking if the EXACT string `Part 1: x = 27` is in the printed output, which it displays as a long string (including explicit newlines as `\n`). If you look, it is true that `Part 1: x = 27` is NOT contained in that string... the reason is, the printout has `part 1: x = 27` where the "p" is not capitalized!!

Here is the output for the section related to the test called `test1_a_printout`

```
_______________________________ test1_a_printout ________________________________

capsys = <_pytest.capture.CaptureFixture object at 0x109622ad0>

    def test1_a_printout(capsys):
        hw01.main()
        captured = capsys.readouterr()
>       assert "Part 1: a = 1.5" in captured.out, "Tip: check the printout for a is EXACT"
E       AssertionError: Tip: check the printout for a is EXACT
E       assert 'Part 1: a = 1.5' in 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPar
E        +  where 'part 1: x = 27\nPart 1: y = 1.0\nPart 1: a =  1.5\nPart 1: b = 7\nPart 1: c = -1\nPart

test_hw01.py:18: AssertionError
```

Here the error is that `Part 1: a = 1.5` is not in the printout. And this happens because `Part 1: a =  1.5` is WRONG!! the reason is: there is an extra space after the equal sign.

After you correct these errors,

```
x=27
y=1.0
a=1.5
b=7
c=-1
result1 = (3*x - 9*y)/(2*a)*(b-c)
print("Part 1: x =",x)
```

```
print("Part 1: y =",y)
print("Part 1: a =",a)
print("Part 1: b =",b)
print("Part 1: c =",c)
print("Part 1: result =", result1)
```

you can see the output change:

```
test_hw01.py::test1_x_printout PASSED
test_hw01.py::test1_y_printout PASSED
test_hw01.py::test1_a_printout PASSED
test_hw01.py::test1_b_printout PASSED
test_hw01.py::test1_c_printout PASSED
test_hw01.py::test1_result1_printout FAILED
test_hw01.py::test2_x_printout FAILED
test_hw01.py::test2_y_printout FAILED
test_hw01.py::test2_result2_printout FAILED
test_hw01.py::test3_a_printout FAILED
test_hw01.py::test3_b_printout FAILED
test_hw01.py::test3_result3_printout FAILED
test_hw01.py::test4_result4_printout FAILED
```

Now you continue adding code and checking the errors for each test until you pass all the tests.

If every test passes

You should see something like this:

```
pfrank@Pablos-MBP hw01 % pytest -s -v
zsh: /usr/local/bin/pytest: bad interpreter: /usr/local/opt/python@3.11/bin/python3.11: no such file or di
====================================== test session starts ======================================
platform darwin -- Python 3.13.1, pytest-8.3.5, pluggy-1.5.0 -- /usr/local/bin/python3
cachedir: .pytest_cache
rootdir: /Users/pfrank/Library/CloudStorage/Dropbox/Pablo/Smith/Academic/Teaching/CSC110/csc110-25f/homewo
collected 13 items

test_hw01.py::test1_x_printout PASSED
test_hw01.py::test1_y_printout PASSED
test_hw01.py::test1_a_printout PASSED
test_hw01.py::test1_b_printout PASSED
test_hw01.py::test1_c_printout PASSED
test_hw01.py::test1_result1_printout PASSED
test_hw01.py::test2_x_printout PASSED
test_hw01.py::test2_y_printout PASSED
test_hw01.py::test2_result2_printout PASSED
test_hw01.py::test3_a_printout PASSED
test_hw01.py::test3_b_printout PASSED
test_hw01.py::test3_result3_printout PASSED
test_hw01.py::test4_result4_printout PASSED

====================================== 13 passed in 0.08s =======================================
```