EECS 1015 Lab 4

Goal

- Be able to write functions
- Be able to write function design recipe
- Learn how to debug functions
- Learn to come up with test cases

Tasks

1. Guide you through the process of writing functions and function design recipe and design test cases
2. Learn to debug function and correct function design recipe and test cases
3. Learn to write function design recipe and the assert statement
4. Learn to write function design recipe and the assert statement
5. Learn to write function design recipe and the assert statement
6. Learn to write function design recipe and the assert statement

Total credit: 100 pts

You are very welcome to ask clarification questions to TAs. But please read the document carefully before you ask questions.

It is an academic offense to copy code from other students, provide code to other students, let other people (including the teaching staff) debug your code, or debug other students' code.

We will check code similarities at the end of the semester.

Questions 3 – 6 may not be arranged by the difficulty level. You are highly recommended to take a look and decide the order of completing the questions. You will experience something similar in the exams and it is important to learn how to triage your tasks.

Task 1: Follow the Steps (30 pts)

In this lab, we will focus on functions and function design recipe. Function design recipe is more than documentation. It provides guidance on how to design your function by carefully designing the test cases and clarifying the types of the function signature. It should be done before writing functions. However, in order to lower your workload, we will revisit the examples in the previous two labs.

Recall the example of calculating fruit prices in the lab 2. You want to buy some fruit in a grocery store and want to know the total price. Here is the price of the fruits available in the store:

- Peaches: $ 3.99 / each
- Apples: $ 2.94 / each
- Watermelons: $ 7.99 / each

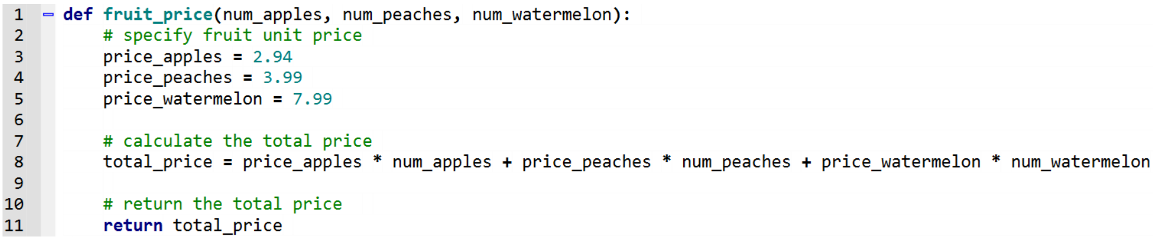
We came up with this piece of code to calculate the total price:

```
# specify fruit quantity
num_apples = int(input("How many apples? "))
num_peaches = int(input("How many peaches? "))
num_watermelon = int(input("How many watermelons? "))

# specify fruit unit price
price_apples = 2.94
price_peaches = 3.99
price_watermelon = 7.99

# calculate the total price
total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

# show the total price
print(total_price)
```

You are highly recommended to attempt the questions yourself before you look at the solution as a way to learn.

Q1.1 Please wrap up the code as a function.

Solution:



```
def fruit_price(num_apples, num_peaches, num_watermelon):
    # specify fruit unit price
    price_apples = 2.94
    price_peaches = 3.99
    price_watermelon = 7.99

    # calculate the total price
    total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

    # return the total price
    return total_price
```

There are 3 key things to decide:

- Function name
- Arguments
- The variable to return

The *function name* should be meaningful and follow the PEP-8 convension (i.e., in the format of lowercases and with words separated by ‘_’).

The *arguments* are usually what value you want from your user. They are straightforward in our examples as they are indicated by the `input()` function. When designing your own function from scratch, you can use the same idea, which is if without using functions, what values do you want to take from your user using `input()` and those should be the parameters.

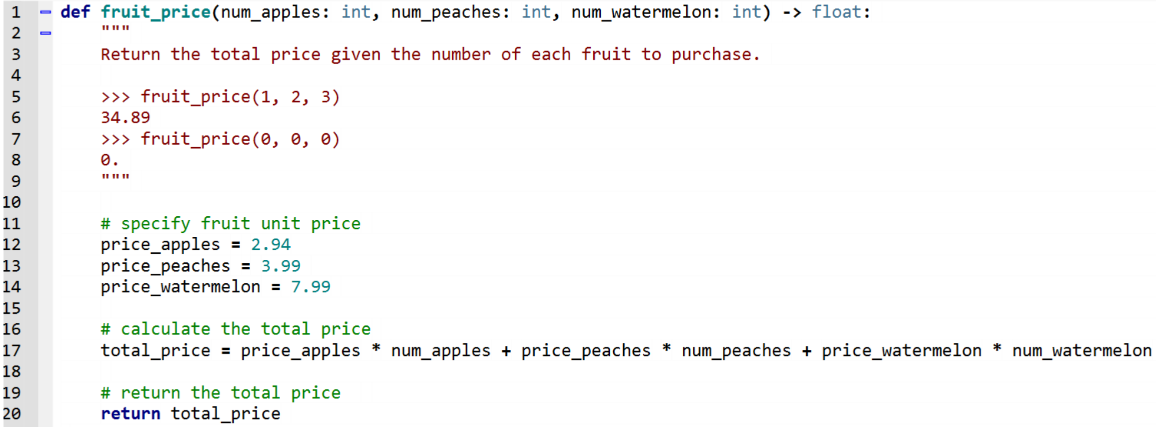
The *variable to return* is the value that you want to obtain at the end. This is indicated by what to print in our examples. If you implement a function from scratch, think about what is the goal of your function and what is the value that you want to obtain at the end.

To convert the code we wrote in the previous labs to functions, you need to complete the following steps:

- Write “def”, then your function name, with a space in between
- Put the arguments in parentheses after the function name
- Remember to put a “:” after the parentheses
- For the rest of code, make sure the code are indented (make sure you remove the lines with `input()`)
- For whatever value that you would like to obtain (which is what you printed in previous labs), write the return keyword and followed by the variable to return, with a space in between

Q1.2 Please write the function design recipe for this function

Solution:



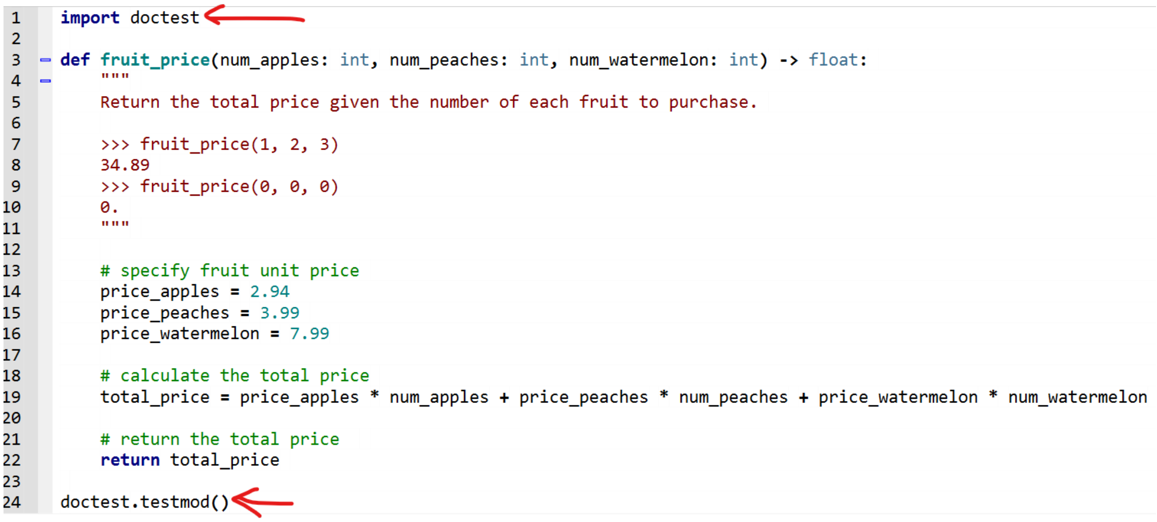
```python
def fruit_price(num_apples: int, num_peaches: int, num_watermelon: int) -> float:
    """
    Return the total price given the number of each fruit to purchase.

    >>> fruit_price(1, 2, 3)
    34.89
    >>> fruit_price(0, 0, 0)
    0.
    """

    # specify fruit unit price
    price_apples = 2.94
    price_peaches = 3.99
    price_watermelon = 7.99

    # calculate the total price
    total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

    # return the total price
    return total_price
```

The function design recipe includes three main components:

- Argument annotation
  - It specifies the data type of each argument, as well as the data type of the return data type
- Description of the function
  - The first in the triple quote provides a human readable description to show the purpose of the function

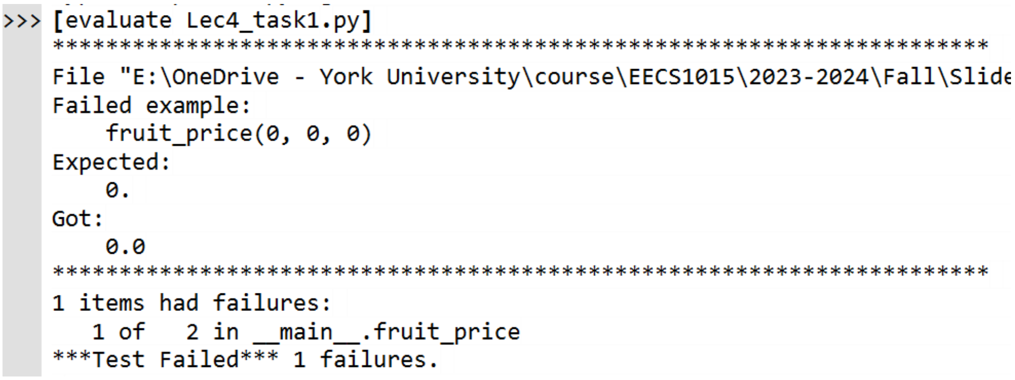
- Test cases
    - The test cases are written to test your function to make sure it works as planned. As mentioned in class, your should have a variety of different cases. In this example, we include a regular case, and a “zero” case.

To run your test cases, you can add the two lines in your code then run your code:



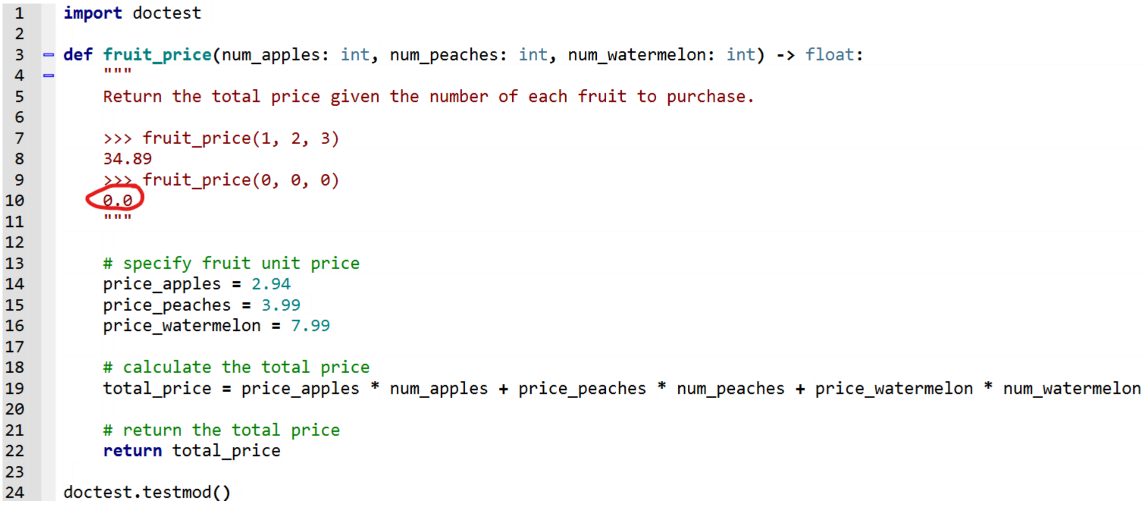
```
import doctest

def fruit_price(num_apples: int, num_peaches: int, num_watermelon: int) -> float:
    """
    Return the total price given the number of each fruit to purchase.

    >>> fruit_price(1, 2, 3)
    34.89
    >>> fruit_price(0, 0, 0)
    0.
    """

    # specify fruit unit price
    price_apples = 2.94
    price_peaches = 3.99
    price_watermelon = 7.99

    # calculate the total price
    total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

    # return the total price
    return total_price

doctest.testmod()
```

Make sure that those two lines are not indented. You will not see any messages if your function passes those test cases. In our case, you will see this message:



```
>>> [evaluate Lec4_task1.py]
    **********************************************************************
    File "E:\OneDrive - York University\course\EECS1015\2023-2024\Fall\Slide
    Failed example:
        fruit_price(0, 0, 0)
    Expected:
        0.
    Got:
        0.0
    **********************************************************************
    1 items had failures:
       1 of   2 in __main__.fruit_price
    ***Test Failed*** 1 failures.
```

You will see that it failed the case where all parameters are 0. (Note: We call it argument when define the function, and call it parameter when make a function call.) This is a limitation of doctest that it compares the result as a string, and they need to be exactly the same, rather than comparing the value. In this case, just modify the test case:



```python
import doctest

def fruit_price(num_apples: int, num_peaches: int, num_watermelon: int) -> float:
    """
    Return the total price given the number of each fruit to purchase.

    >>> fruit_price(1, 2, 3)
    34.89
    >>> fruit_price(0, 0, 0)
    0.0
    """

    # specify fruit unit price
    price_apples = 2.94
    price_peaches = 3.99
    price_watermelon = 7.99

    # calculate the total price
    total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

    # return the total price
    return total_price

doctest.testmod()
```

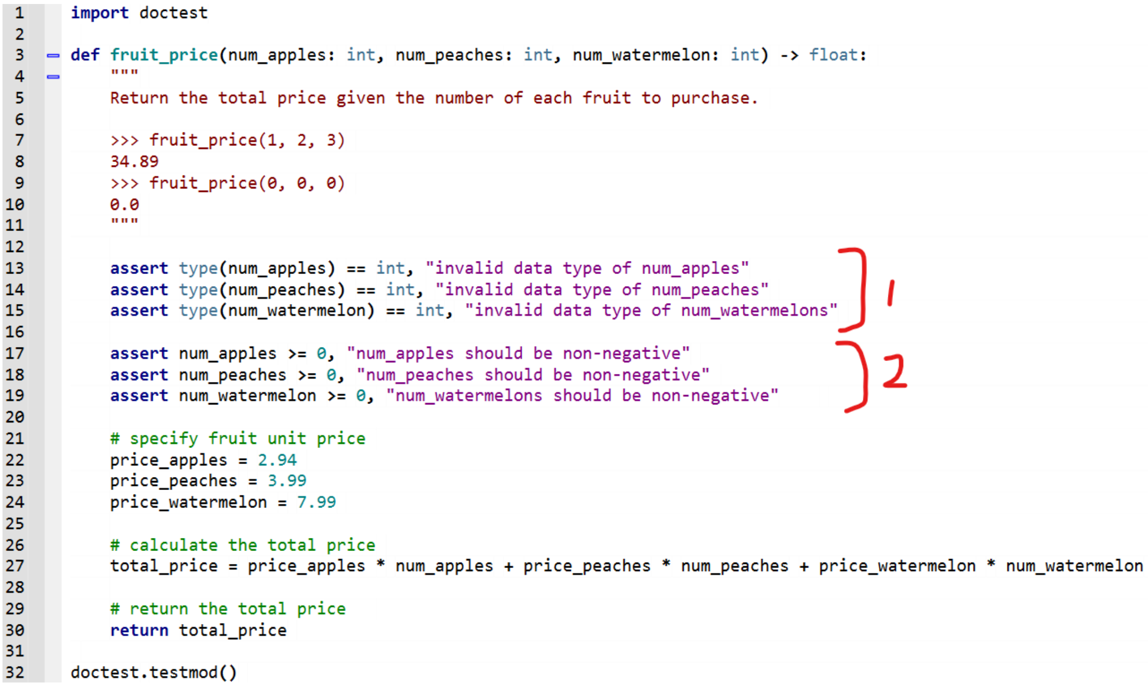
Then your function should pass all test cases and do not print anything when run the script.

Q1.3 Add code to check the precondition.

Solution:

Between you write the code, think about what the precondition should be. The basic constraint is that it should be with the correct data type. For this specific example, all arguments should be integers. Is there any other constraint?

Since the arguments are the number of fruits, it cannot take negative numbers. So another constraint is that they should be non-negative. Let's add code to check the precondition:



```python
import doctest

def fruit_price(num_apples: int, num_peaches: int, num_watermelon: int) -> float:
    """
    Return the total price given the number of each fruit to purchase.

    >>> fruit_price(1, 2, 3)
    34.89
    >>> fruit_price(0, 0, 0)
    0.0
    """

    assert type(num_apples) == int, "invalid data type of num_apples"
    assert type(num_peaches) == int, "invalid data type of num_peaches"
    assert type(num_watermelon) == int, "invalid data type of num_watermelons"

    assert num_apples >= 0, "num_apples should be non-negative"
    assert num_peaches >= 0, "num_peaches should be non-negative"
    assert num_watermelon >= 0, "num_watermelons should be non-negative"

    # specify fruit unit price
    price_apples = 2.94
    price_peaches = 3.99
    price_watermelon = 7.99

    # calculate the total price
    total_price = price_apples * num_apples + price_peaches * num_peaches + price_watermelon * num_watermelon

    # return the total price
    return total_price

doctest.testmod()
```

Where the first chunk of code to check the data type and the second chunk of code for additional constraints.

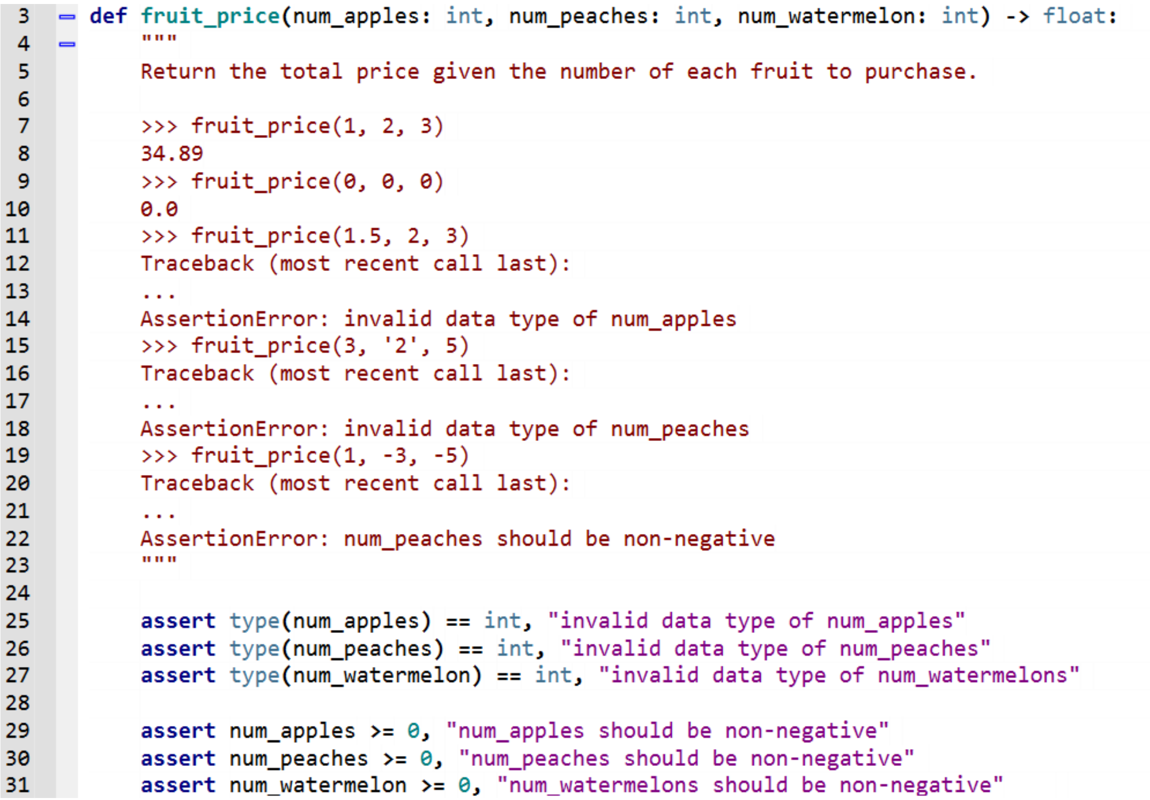
The syntax of the `assert` is that it should satisfy the expression right after it (e.g., the Boolean expression should be `True`), otherwise it will print an error message with the information after the “,”

Note: `type()` returns the type of a variable. To check whether a variable is with certain type, just call the type function and see whether it equal equal to a particular type (e.g., `type(num_apples) == int`). You can also check other types, such as check whether something is a string (e.g., `type(num_apples) == str`) or a float (e.g., `type(num_apples) == float`).

Q1.4 Write relevant test cases to check the preconditions.

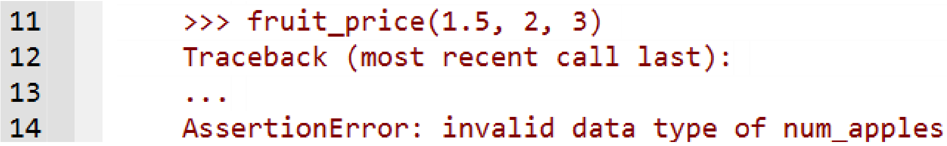
Solution:

(Due to space limit, the rest of the code is omitted as they are the same as the previous image)



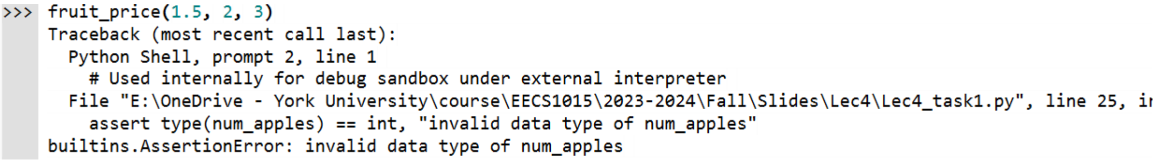
```python
def fruit_price(num_apples: int, num_peaches: int, num_watermelon: int) -> float:
    """
    Return the total price given the number of each fruit to purchase.

    >>> fruit_price(1, 2, 3)
    34.89
    >>> fruit_price(0, 0, 0)
    0.0
    >>> fruit_price(1.5, 2, 3)
    Traceback (most recent call last):
    ...
    AssertionError: invalid data type of num_apples
    >>> fruit_price(3, '2', 5)
    Traceback (most recent call last):
    ...
    AssertionError: invalid data type of num_peaches
    >>> fruit_price(1, -3, -5)
    Traceback (most recent call last):
    ...
    AssertionError: num_peaches should be non-negative
    """

    assert type(num_apples) == int, "invalid data type of num_apples"
    assert type(num_peaches) == int, "invalid data type of num_peaches"
    assert type(num_watermelon) == int, "invalid data type of num_watermelons"

    assert num_apples >= 0, "num_apples should be non-negative"
    assert num_peaches >= 0, "num_peaches should be non-negative"
    assert num_watermelon >= 0, "num_watermelons should be non-negative"
```

You’ll notice that instead of showing the value of a data, the text expects a lot of texts, such as:



```
    >>> fruit_price(1.5, 2, 3)
    Traceback (most recent call last):
    ...
    AssertionError: invalid data type of num_apples
```

This is because instead of returning a value, your function will throw an error message. If you run the script (i.e., Python Shell -> Options -> Evaluate lab4_task1.py, do not use the run button with the green triangle), and if you run the function with the parameters, the console will show:



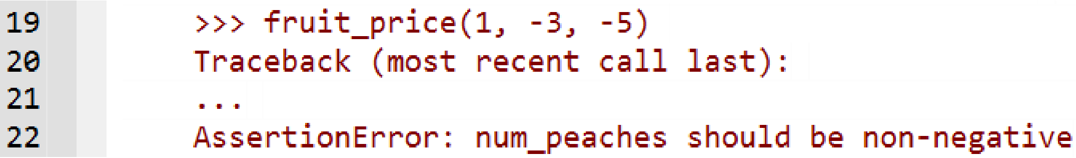
```
>>> fruit_price(1.5, 2, 3)
    Traceback (most recent call last):
      Python Shell, prompt 2, line 1
        # Used internally for debug sandbox under external interpreter
      File "E:\OneDrive - York University\course\EECS1015\2023-2024\Fall\Slides\Lec4\Lec4_task1.py", line 25, i
        assert type(num_apples) == int, "invalid data type of num_apples"
    builtins.AssertionError: invalid data type of num_apples
```

It has a lot more texts than the output in the test case. We omit them with "…" on purpose. Note that in the third line from the bottom in the image above, it shows "File "E:\OneDrive - York University\course\EECS1015\2023-2024\Fall\…". This is the location of the python script on my computer. However, you may store the file at a different location on your computer. So if we simply copy the error message, you will need to modify the test case to comply with the actual location on your computer, which is very inconvenient. Therefore, when writing the output for the test cases that expect to show error messages, we do the following:

- In the first line, write `Traceback (most recent call last):`
- In the second line, write `...`
- In the third line, start with `AssertionError:` , this is because the error is reported from your `assert` statement, so it should be an assertionError
- Then copy the error message you have in your assert statement after the ":"
- Make sure that there is a space between ":" and the error message.
- Make sure your test case is indented properly.

One common mistake is the trailing spaces after the texts in the output of your texts. Recall that doctest will compare the strings as is, so if you have trailing spaces in the third line, then it will compare the string `"AssertionError: invalid data type of num_apples      "` with the actual string `"AssertionError: invalid data type of num_apples"` which has no space at the end. Then the test case will report failure. The spaces are very likely to be added when you copy and paste texts.

Also in our last test case:



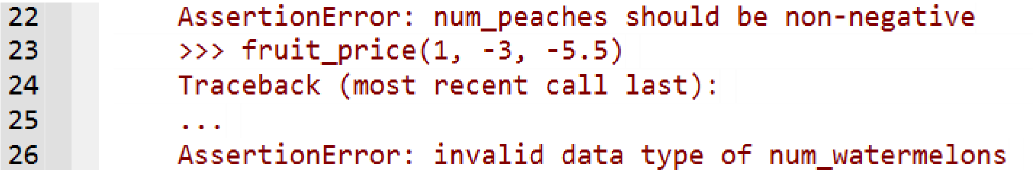
```
    >>> fruit_price(1, -3, -5)
    Traceback (most recent call last):
    ...
    AssertionError: num_peaches should be non-negative
```

Although both the second and the third parameter are negative, it only complains about the second one, but not the third one. This is because it will stop at the first `assert` statement when it failed the test, and the code will not proceed.

Q5. Based on the information, think about what output you should write if you want to write a test case for `fruit_price(1, -3, -5.5)`.

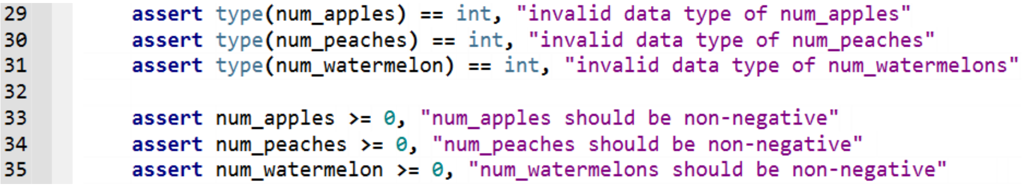
Solution:

It should looks like:



```
    AssertionError: num_peaches should be non-negative
    >>> fruit_price(1, -3, -5.5)
    Traceback (most recent call last):
    ...
    AssertionError: invalid data type of num_watermelons
```

This is because our assert statement is written in the following order:



```
29      assert type(num_apples) == int, "invalid data type of num_apples"
30      assert type(num_peaches) == int, "invalid data type of num_peaches"
31      assert type(num_watermelon) == int, "invalid data type of num_watermelons"
32
33      assert num_apples >= 0, "num_apples should be non-negative"
34      assert num_peaches >= 0, "num_peaches should be non-negative"
35      assert num_watermelon >= 0, "num_watermelons should be non-negative"
```

The `assert` statement in line 31 will throw an error message, so it will stop and will not check the statements in line 33 – 35.

Submission

- Submit your work on PrairieLearn
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need spend some time debugging!).

Rubrics:

- You will get full mark for this question if you submit the solution (with no typo) to the required location on Prairielearn before the deadline.

Note:

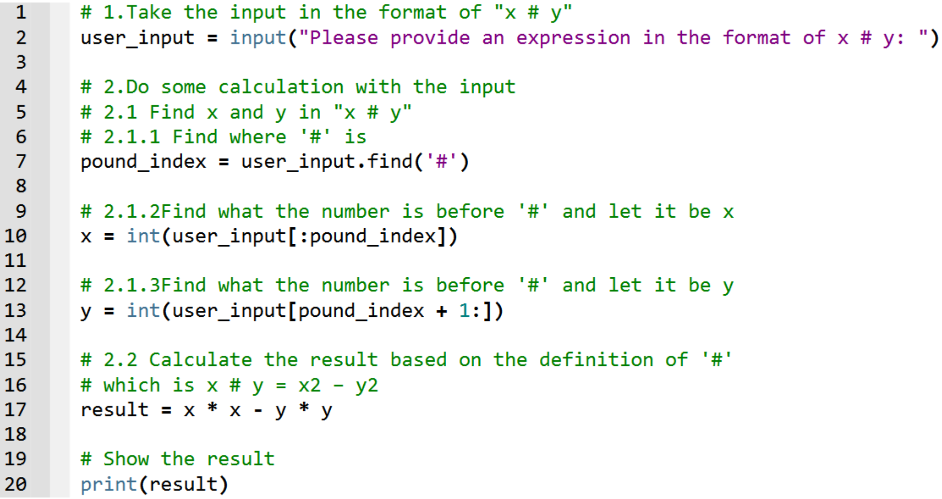
- since we provide solutions to this question, it is ok to submit it as is.
- In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments
- The autograder may take some time to grade

Task 2: Debugging (30 pts)

Let's revisit task 1 in lab 3. we define a new operation # between two natural numbers x # y such that it results in $x^2 - y^2$. For example:

$$2 \# 3 = 2^2 – 3^2 = 4 – 9 = -5$$

The code we had was:

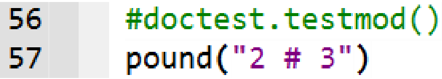


```
# 1.Take the input in the format of "x # y"
user_input = input("Please provide an expression in the format of x # y: ")

# 2.Do some calculation with the input
# 2.1 Find x and y in "x # y"
# 2.1.1 Find where '#' is
pound_index = user_input.find('#')

# 2.1.2Find what the number is before '#' and let it be x
x = int(user_input[:pound_index])

# 2.1.3Find what the number is before '#' and let it be y
y = int(user_input[pound_index + 1:])

# 2.2 Calculate the result based on the definition of '#'
# which is x # y = x2 - y2
result = x * x - y * y

# Show the result
print(result)
```

Let's convert the code to a function and add the function design recipe. The code is a bit too long for a screenshot, please see the code in lab4_task2.py

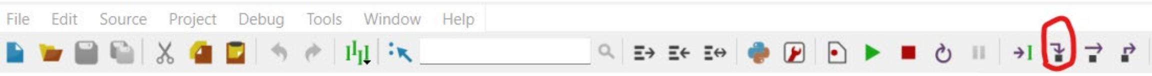
The function should be called pound, and should return an `int`. The code to debug (lab4_task2.py) may have syntax errors, run-time errors, semantic errors or error in test cases. To help with debugging, you can use the debugger to execute the code line by line (you should fix the syntax errors before doing this).

To debug code for a specific example, you can put the code at the end of the file and run the debugger. When doing this, you can comment out the line with `doctest.testmod()`



```
56  #doctest.testmod()
57  pound("2 # 3")
```

When you are debugging you code and reach line 57, since you want to step into this function, you need to click this button:



Requirement

- You only need to fix the code to pass the test cases specified in the document. Note that the code is still not perfect and do not check for cases such as "2 ## 3". You can think about how to improve it and you will work on it in task 6.

Submission

Submit the Python code on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need spend some time debugging!).

Note:

- In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive points if there are syntax errors in the code
- Otherwise
  - You will receive 10 pts for including the function description
  - You will receive 10 pts for correctly specifying the argument annotation
  - You will receive 4 pts for passing all 4 test cases included in the starter code
  - You will receive 6 pts for passing additional tests

Task 3: Implementation (10 pts)

In this task, you will implement a function and write design recipe by expanding lab4_task3.py . It revisit lab 2 task 2.

The problem is to count the total number of wheels given different numbers of vehicles:

- A bicycle: 2 wheels
- A tricycle: 3 wheels
- A car: 4 wheels
- A huge truck: 18 wheels

For example, given 1 bicycle, 0 tricycle, 2 cars and 5 trucks, there are in total:

2 x 1 + 3 x 0 + 4 x 2 + 18 x 5 = 100 wheels

Requirement

- The function name must be: `count_wheels`
- You must NOT change the order of the arguments
- Please remove on "pass" line 2. It is a keyword simply to keep it free of syntax errors

Submission

Submit the Python code on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 2 pts if you provide a description to the function
  - You will receive 3 pts if you provide the argument annotations correctly
  - You will receive 4 pts if you include at least 4 test cases and your code pass those test cases
  - You will receive 1 pt if your function passes additional test cases

Task 4: Implementation (10 pts)

In this task, you will implement a function and write design recipe by expanding lab4_task4.py . It revisits lab 3 task 3.

The problem is to determine whether a child should buy a ticket to enter a park. A child under the age of 6 (including 6) does not need to buy a ticket, but if the child is taller than 120 cm (not including 120), the child should buy a ticket regardless of the age.

If a child needs to buy a ticket, your function should return True, otherwise show False.

Hint: This question is a little trickier because you can provide either int or float as the parameters. For the argument annotations, you can assume that it is a float as it is a more general. But when checking the preconditions in your function body, you may need to be careful and consider both int and float.

Requirement

- The function name must be: `need_to_buy_ticket`
- You must NOT change the order of the arguments
- Please remove on "pass" line 2. It is a keyword simply to keep it free of syntax errors

Submission

Submit the Python code on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 2 pts if you provide a description to the function
  - You will receive 3 pts if you provide the argument annotations correctly
  - You will receive 4 pts if you include at least 4 test cases and your code pass those test cases
  - You will receive 1 pt if your function passes additional test cases

Task 5: Implementation (10 pts)

In this task, you will implement a function and no starter code is provided. In this task, let's revisit the `xor` logical operator. Recall that this operator results in `True` if and only if one of the boolean variables is `True`.

Requirement

- The function name must be: `xor_operator`
- The function takes two parameters and both are Boolean variables
- The function returns the boolean result

Submission

Submit the Python code on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need to spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 1 pts if you provide a description to the function
  - You will receive 2 pts if you provide the argument annotations correctly
  - You will receive 2 pts if you include at least 4 test cases and your code pass those test cases
  - You will receive 5 pt if your function passes additional test cases

Task 6: Implementation (10 pts)

In this task, you will implement a function and no starter code is provided. In this task, let's revisit the pound operator. As we talked about in class, the user may not provide a valid input. For example, the user may provide a string such as `"2 ## 3"`, or `"2 # "`. A user may also provide valid input but with leading or trailing spaces, such as `"       2 # 3"`, or `"2 # 3   "`. In this task, try to add more assert statements to raise errors if the input is invalid, or accept the input with more spaces.

This task is indeed somewhat challenging, but please take it as an opportunity to practice your problem-solving skills and learn to handle a slightly challenging programming question. It is highly recommended that you analyze the question (e.g., how would you solve the question if you are given such as a string) and break it down to smaller pieces BEFORE you start implementation.

Requirement

- The function name must be: pound
- The function takes one parameter and it is a string
- The function returns the calculated result, which is an `int`

Submission

Submit the Python code on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 0.5 pts if you provide a description to the function
  - You will receive 0.5 pts if you provide the argument annotations correctly
  - You will receive 1 pts if you include at least 4 test cases and your code pass those test cases
  - Here are the test cases:
    1. You will receive 1 pt if your function returns the correct result with the input with leading spaces (e.g., `"       2 # 3"`)
    2. You will receive 1 pt if your function returns the correct result with the input with trailing spaces (e.g., `"2 # 3          "`)
    3. You will receive 1 pt if your function returns the correct result with the input with additional spaces (e.g., `"2   #       3"`)
    4. You will receive 1 pt if your function returns the correct result with the input with no spaces (e.g., `"2#3"`)
    5. You will receive 1 pt if your function raised an error with the input with multiple pound operations between numbers (e.g., `"2 ##### 3"`)

6. You will receive 1 pt if your function raised an error with the input with pound operations between multiple numbers (e.g., `"2 # 3 # 4 # 5"`)
7. You will receive 1 pt if your function raised an error with the input with no first number or second number (e.g., `" # 3"`)
8. You will receive 1 pt if your function raised an error with the input if it include floating numbers (e.g., `"2.5 # 3"`)

You are highly recommended to try to convert other tasks in previous labs to functions and write function design recipe. In addition, think about how to handle the cases if the user did not provide valid inputs. You are very welcome to post your thoughts and solutions on the discussion forum.