CS 101 - Line Simulator
Spring 2017
Algorithm Due : **April 30th, 2017**
Program Due : **May 7th, 2017**
All work submitted must be your own.
Deliverables : You only need to submit your solution. You must use functions to modularize your work. You should use exception handling where necessary as well.

Waiting Lines

Programs are useful to simulate real world activities to analyze them and to try other ideas to see which one is better. We're going to work out a simulation of grocery store checkout lines. If we have 4 cash registers, is it better to form one long line, or form 4 lines for each register and not allow people to leave one to get in another.

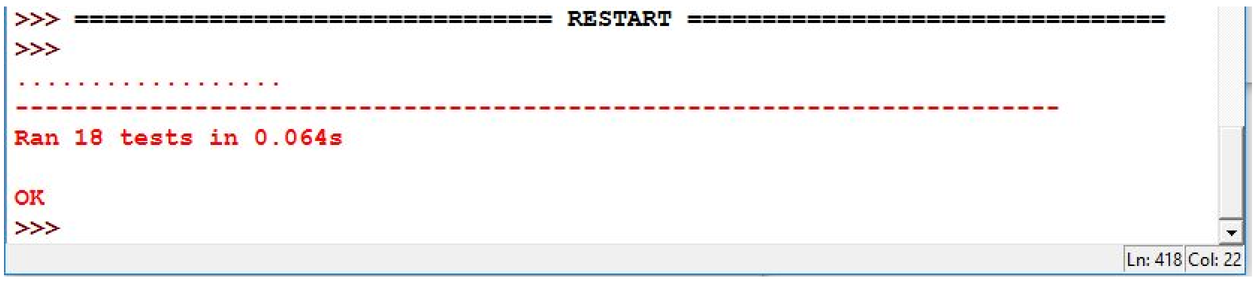
How do we measure what is a better solution?

How will we measure which one is better? We can come up with many different metrics, but we will settle trying to minimize the average wait time of each customer. Next we need to think about what classes we want to implement. It would make sense to have a customer class that has information about when they got into a line, when the got to the register, when they left the register. We can then figure out how long that person waiting by knowing the time they get into the line and get to the register. In order to make our simulation easier we will use an integer for time and just keep an integer as the count of the time. We'll start at clock 1 and then it will be clock 2, etc. Each customer will have the number of items they are buying, we will assume it takes one unit of time to scan and bag each item the person wants. We'll also have a cashier that has information about themselves, it has a line (or list of customers that are waiting to be helped). It's current client.

We've given you a Program8_solution.py file for you to modify and make work. It has the classes stubbed out and defined with descriptions of what has to be completed. You've also been given a Checkout_Testing.py file that has unit tests that must pass in order to consider your program complete. This is not uncommon as an employer will give you unit tests that your code has to pass. These unit tests can help you write finish writing the classes as specified. The Program8_Solution.py file must have its name change since Checkout_Testing.py file imports the file to test.

NOTE : When you make a change to the solution, you must save those changes before running the tests. Make sure you do this.

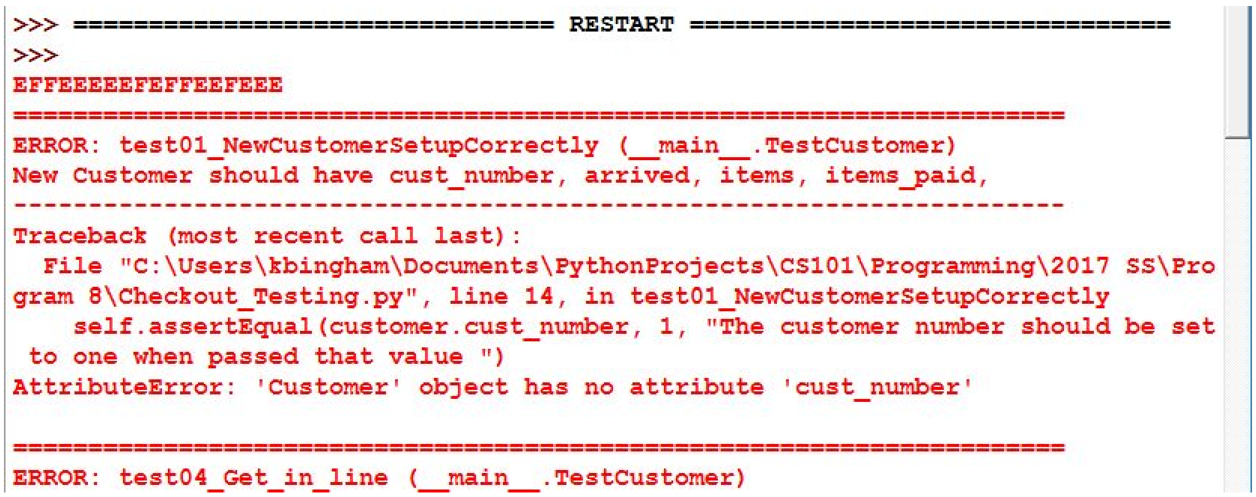
Once you've written all the code for the classes in the solution all of the tests will pass, and will look like this.



```
>>> ================================ RESTART ================================
>>>
..................
----------------------------------------------------------------------
Ran 18 tests in 0.064s

OK
>>>
```

Those line of dots are the tests, and all 18 of them ran successfully.   This means your code runs as given. (it doesn't mean you have a final program).

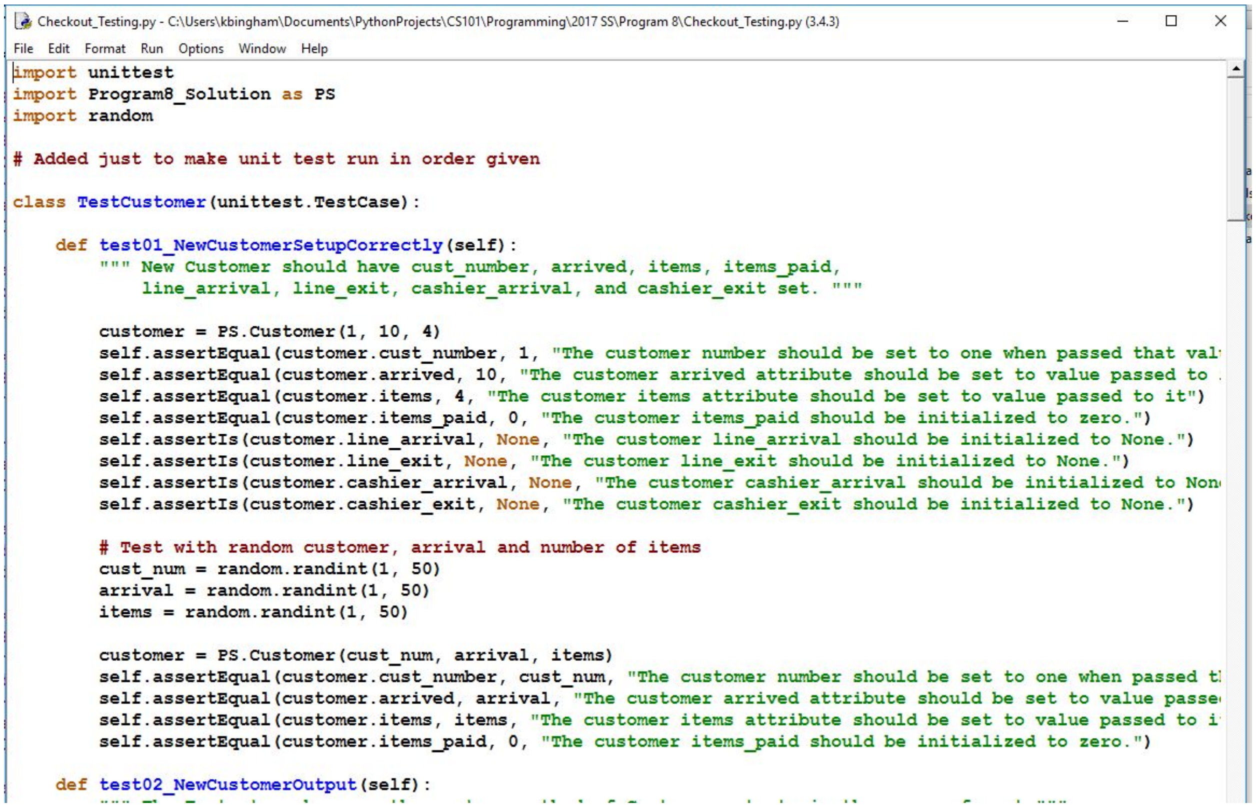
Initially however, all of the tests will give you errors.



```
>>> ================================ RESTART ================================
>>>
EFFEEEEEFEFFEEFEEE
======================================================================
ERROR: test01_NewCustomerSetupCorrectly (__main__.TestCustomer)
New Customer should have cust_number, arrived, items, items_paid,
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\kbingham\Documents\PythonProjects\CS101\Programming\2017 SS\Pro
gram 8\Checkout_Testing.py", line 14, in test01_NewCustomerSetupCorrectly
    self.assertEqual(customer.cust_number, 1, "The customer number should be set
 to one when passed that value ")
AttributeError: 'Customer' object has no attribute 'cust_number'

======================================================================
ERROR: test04_Get_in_line (__main__.TestCustomer)
```

Those are all errors and failures. What you want to do, is read the tests and errors and figure out how to fix it in the solution. Make that change and run the tests again. Slowly you'll be able to get all the tests to pass and you know you've gotten the classes mostly written.

You may want to start by looking at the testing file in order. Read the test and make sure you understand what it is trying to test. Then make the classes work with that test.



Checkout_Testing.py - C:\Users\kbingham\Documents\PythonProjects\CS101\Programming\2017 SS\Program 8\Checkout_Testing.py (3.4.3)

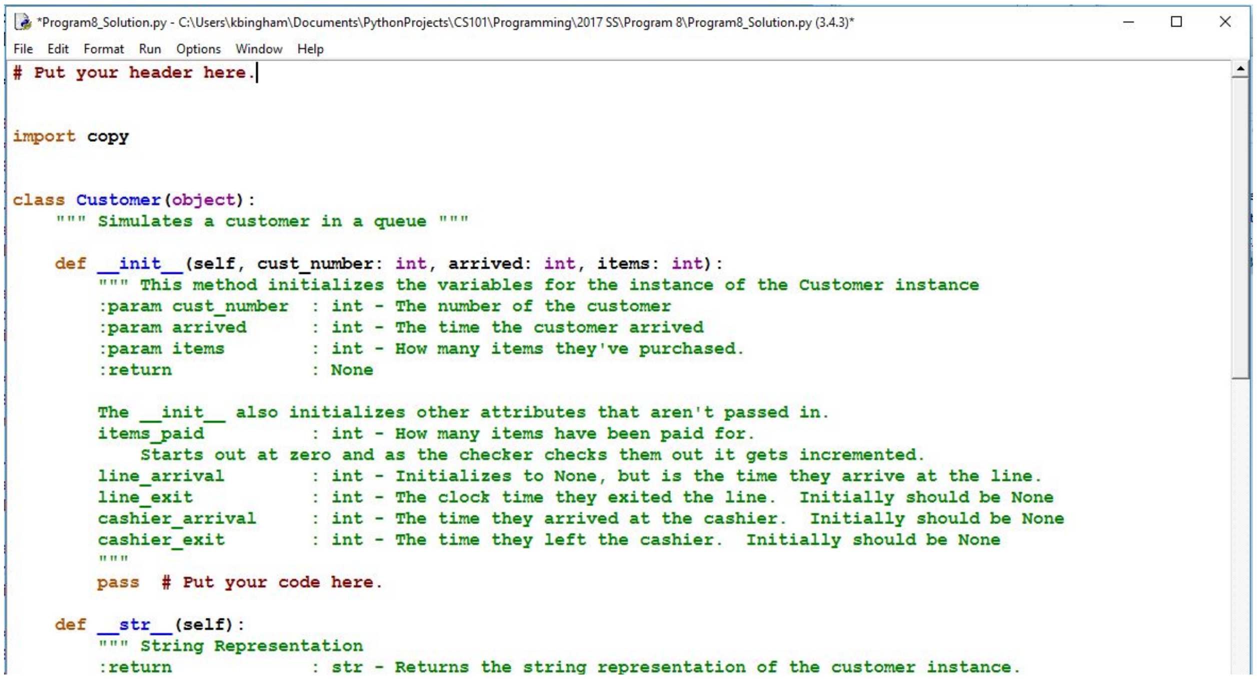
File Edit Format Run Options Window Help

```
import unittest
import Program8_Solution as PS
import random

# Added just to make unit test run in order given

class TestCustomer(unittest.TestCase):

    def test01_NewCustomerSetupCorrectly(self):
        """ New Customer should have cust_number, arrived, items, items_paid,
            line_arrival, line_exit, cashier_arrival, and cashier_exit set. """

        customer = PS.Customer(1, 10, 4)
        self.assertEqual(customer.cust_number, 1, "The customer number should be set to one when passed that val
        self.assertEqual(customer.arrived, 10, "The customer arrived attribute should be set to value passed to 
        self.assertEqual(customer.items, 4, "The customer items attribute should be set to value passed to it")
        self.assertEqual(customer.items_paid, 0, "The customer items_paid should be initialized to zero.")
        self.assertIs(customer.line_arrival, None, "The customer line_arrival should be initialized to None.")
        self.assertIs(customer.line_exit, None, "The customer line_exit should be initialized to None.")
        self.assertIs(customer.cashier_arrival, None, "The customer cashier_arrival should be initialized to Non
        self.assertIs(customer.cashier_exit, None, "The customer cashier_exit should be initialized to None.")

        # Test with random customer, arrival and number of items
        cust_num = random.randint(1, 50)
        arrival = random.randint(1, 50)
        items = random.randint(1, 50)

        customer = PS.Customer(cust_num, arrival, items)
        self.assertEqual(customer.cust_number, cust_num, "The customer number should be set to one when passed t
        self.assertEqual(customer.arrived, arrival, "The customer arrived attribute should be set to value passe
        self.assertEqual(customer.items, items, "The customer items attribute should be set to value passed to i
        self.assertEqual(customer.items_paid, 0, "The customer items_paid should be initialized to zero.")

    def test02_NewCustomerOutput(self):
```

In this one, when a new Customer is created from the solution it expects the cust_number, arrived, items, items_paid, line_arrival, line_exit, cashier_arrival, and cashier_exit attributes to be set correctly.

Most of the methods that are given in the solution are stubbed out, and will only require a few lines of code to complete. So think clearly about what the test is trying to do.

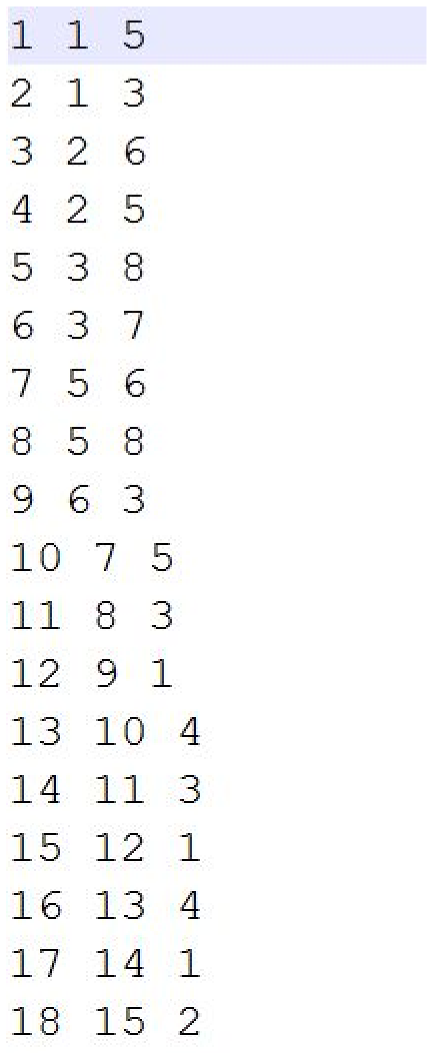
Program 8 Solution


Program8_Solution.py - C:\Users\kbingham\Documents\PythonProjects\CS101\Programming\2017 SS\Program 8\Program8_Solution.py (3.4.3)
File Edit Format Run Options Window Help

```python
# Put your header here.


import copy


class Customer(object):
    """ Simulates a customer in a queue """

    def __init__(self, cust_number: int, arrived: int, items: int):
        """ This method initializes the variables for the instance of the Customer instance
        :param cust_number  : int - The number of the customer
        :param arrived      : int - The time the customer arrived
        :param items        : int - How many items they've purchased.
        :return             : None

        The __init__ also initializes other attributes that aren't passed in.
        items_paid          : int - How many items have been paid for.
            Starts out at zero and as the checker checks them out it gets incremented.
        line_arrival        : int - Initializes to None, but is the time they arrive at the line.
        line_exit           : int - The clock time they exited the line.  Initially should be None
        cashier_arrival     : int - The time they arrived at the cashier.  Initially should be None
        cashier_exit        : int - The time they left the cashier.  Initially should be None
        """
        pass  # Put your code here.

    def __str__(self):
        """ String Representation
        :return             : str - Returns the string representation of the customer instance.
```

You'll notice all the methods have been created for you. You simply need to write the code to make it work correctly.

Once you have the classes, you'll then have to write a program that will use those classes to simulate a line. We will read in information from a file called arrivals.txt. Remember you should use proper error handling when opening files. This file contains the customer number and when they come to the line (the clock) and how many items they have.



```
1 1 5
2 1 3
3 2 6
4 2 5
5 3 8
6 3 7
7 5 6
8 5 8
9 6 3
10 7 5
11 8 3
12 9 1
13 10 4
14 11 3
15 12 1
16 13 4
17 14 1
18 15 2
```

We want to try the simulation with 4 cashiers and 4 separate lines. The world will increment the clock count and our simulation will program. So at clock count 2, both customers 1 and 2 above go 2 open registers since no one is waiting. At clock cycle 2, customers 1 and 2 have 1 items paid for and still have multiple items to finish out. Customers 3, and 4 enter the lines and go to the 2 available registers. At clock 3, customers 5, and 6 go the line, but all the registers are being used so they both choose the shortest line they can, and begin waiting., etc etc.

Below I show the sample output for 4 separate lines, one for each register.

Sample Program

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>> 

The clock is at 1
Checker $(1) - C(1) <-
Checker $(2) - C(2) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 2
Checker $(1) - C(1) <-
Checker $(2) - C(2) <-
Checker $(3) - C(3) <-
Checker $(4) - C(4) <-

The clock is at 3
Checker $(1) - C(1) <- C(5)
Checker $(2) - C(2) <- C(6)
Checker $(3) - C(3) <-
Checker $(4) - C(4) <-

The clock is at 4
Checker $(1) - C(1) <- C(5)
Checker $(2) - C(6) <-
Checker $(3) - C(3) <-
Checker $(4) - C(4) <-

The clock is at 5
Checker $(1) - C(1) <- C(5)
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(3) <- C(8)
Checker $(4) - C(4) <-

The clock is at 6
Checker $(1) - C(5) <-
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(3) <- C(8)
Checker $(4) - C(4) <- C(9)

The clock is at 7
Checker $(1) - C(5) <- C(10)
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(3) <- C(8)
Checker $(4) - C(9) <-

The clock is at 8
Checker $(1) - C(5) <- C(10)
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(8) <-
Checker $(4) - C(9) <- C(11)

The clock is at 9
Checker $(1) - C(5) <- C(10)
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(9) <- C(11)

The clock is at 10
Checker $(1) - C(5) <- C(10),C(13)
Checker $(2) - C(6) <- C(7)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(11) <-

The clock is at 11
```

```
Checker $(1) - C(5) <- C(10),C(13)
Checker $(2) - C(7) <-
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(11) <- C(14)

The clock is at 12
Checker $(1) - C(5) <- C(10),C(13)
Checker $(2) - C(7) <- C(15)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(11) <- C(14)

The clock is at 13
Checker $(1) - C(5) <- C(10),C(13)
Checker $(2) - C(7) <- C(15),C(16)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(14) <-

The clock is at 14
Checker $(1) - C(10) <- C(13)
Checker $(2) - C(7) <- C(15),C(16)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(14) <- C(17)

The clock is at 15
Checker $(1) - C(10) <- C(13),C(18)
Checker $(2) - C(7) <- C(15),C(16)
Checker $(3) - C(8) <- C(12)
Checker $(4) - C(14) <- C(17)

The clock is at 16
Checker $(1) - C(10) <- C(13),C(18)
Checker $(2) - C(7) <- C(15),C(16)
Checker $(3) - C(12) <-
Checker $(4) - C(17) <-

The clock is at 17
Checker $(1) - C(10) <- C(13),C(18)
Checker $(2) - C(15) <- C(16)
Checker $(3) <-
Checker $(4) <-

The clock is at 18
Checker $(1) - C(10) <- C(13),C(18)
Checker $(2) - C(16) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 19
Checker $(1) - C(13) <- C(18)
Checker $(2) - C(16) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 20
Checker $(1) - C(13) <- C(18)
Checker $(2) - C(16) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 21
Checker $(1) - C(13) <- C(18)
Checker $(2) - C(16) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 22
Checker $(1) - C(13) <- C(18)
Checker $(2) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 23
Checker $(1) - C(18) <-
Checker $(2) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 24
Checker $(1) - C(18) <-
Checker $(2) <-
Checker $(3) <-
Checker $(4) <-

The clock is at 25
Checker $(1) <-
Checker $(2) <-
Checker $(3) <-
Checker $(4) <-

The average wait time is  << information redacted >>

>>>
```

I've redacted the information about the average wait time. You'll be expected to show it.

Specification

- The arrivals.txt file will be the same format, but may have different data for testing.
- Read through the txt file and simulate a system with 4 registers a line for each, and a system with one line that feeds all 4 registers to determine which is quicker.

Point Breakdown - May be modified as needed

Points	**Requirement**
5	Header
10	Readability, variable naming, comments

5	Data Structures
5	Proper functions
15	Create proper classes and pass the unit tests
2	Read the arrivals.txt
2	Simulate the world clock and customers entering the line
8	Calculate the average wait time for 4 lines
8	Calculate the average wait time for 1 line feeding all 4 registers

30 points off for programs that crash on expected input.

References

1. Unit Testing - https://docs.python.org/3.6/library/unittest.html