Problem Set 1

Handed out: Friday, September 8, 2017.

Due: 5:00 PM, Friday, September 15, 2017

This problem set will introduce you to using control flow in Python and formulating a computational solution to a problem. It will also give you a chance to explore bisection search. This problem set has **three** problems. You should save your code for the first problem as **ps1a.py**, the second problem as **ps1b.py** and the third problem as **ps1c.py**, and make sure to hand in all three files. **If you do not name these files as directed, you will receive a zero on this problem set because our grader will not run properly**. Don't forget to include comments to help us understand your code! In the zip file for this pset, in addition to this pdf, are two other files. One is a tester file for you to run and test your code with, and you will be told to uncomment lines to aid in testing throughout the pset. The other is a helper file that allows us to test your code - do not change the contents of this file!

Collaboration

You may work with other students; however, each student should write up and hand in his or her assignment separately. Be sure to indicate with whom you have worked in a comment at the start of each file.

Before You Start:

Read the style guide sections 1, 2, and 3.

In order for your program to be graded correctly, **you must declare your variables as the first lines of the program**. The **order** in which to declare the variables and the **variable names** that must be used are in the notes section of each part. **Failure to do so will result in a 0 on the problem set.**

Part A: House Hunting

You have graduated from MIT and now have a great job! You move to the San Francisco Bay Area and decide that you want to start saving to buy a house. As housing prices are very high in the Bay Area, you realize you are going to have to save for several years before you can afford to make the down payment on a house. In Part A, we are going to determine how many months it will take you to save enough money to make the down payment given the following information:

1. Call your annual salary **annual_salary**.
2. Assume you are going to dedicate a certain amount of your salary each month to saving for the down payment. Call that **portion_saved**. This variable should be in

decimal form (i.e. 0.1 for 10%).
3. Call the cost of your dream home **total_cost**.
4. Call the portion of the cost needed for a down payment **portion_down_payment**. For simplicity, assume that portion_down_payment = 0.18 (18%).
5. Call the amount that you have saved thus far **current_savings**. You start with a current savings of $0.
6. Assume that you invest your current savings wisely, with an annual return of **r** (in other words, at the end of each month, you receive an additional current_savings*r/12 funds to put into your savings – the 12 is because **r** is an annual rate). Assume that your investments earn a return of r = 0.03 (3%).
7. "At the end of each month you will add a percentage of your monthly salary to your savings. You will also gain a monthly return on your investment (Note: investment here is your savings before the end of this month)"
8. Call the number of months required **months**.

Write a program to calculate how many months it will take you to save up enough money for a down payment. You will want your main variables to be floats, so you should cast user inputs to floats.

Your program should ask the user to enter the following variables:
1. The starting annual salary (**annual_salary**)
2. The portion of salary to be saved (**portion_saved**)
3. The cost of your dream home (**total_cost**)

Notes

- Initialize variables with the names given in bold. These variables must be initialized in the following order, at the beginning of your program, before declaring any other variables:
  1. **annual_salary**
  2. **portion_saved**
  3. **total_cost**

  You can add additional variables as needed, but they must come after those listed in bold. Be careful about values that represent annual amounts and those that represent monthly amounts.
    - If the variables are not initialized to exactly the names above and as the first few lines, the problem set will be graded incorrectly
- Look at input() if you need help with getting user input. For this problem set, you can assume that users will enter valid input (e.g. they won't enter a string when you expect an int)

Try different inputs and see how long it takes to save for a down payment. **Please make your program print results in the format shown in the test cases below.**

Test Case 1

```
>>>
Enter your annual salary: 110000
Enter the percent of your salary to save, as a decimal: .15
Enter the cost of your dream home: 750000
```

```
Number of months: 88
>>>
```

Test Case 2

```
>>>
Enter your annual salary: 65000
Enter the percent of your salary to save, as a decimal: .20
Enter the cost of your dream home: 400000
Number of months: 62
>>>
```

Test Case 3

```
>>>
Enter your annual salary: 350000
Enter the percent of your salary to save, as a decimal: .3
Enter the cost of your dream home: 10000000
Number of months: 167
>>>
```

Additionally, open the file ps1_tester.py, and uncomment lines 5 and 6 by erasing the # at the beginning of the line. When you run this file, you should pass the first three test cases. In order for this to work successfully, ps1a.py must be in the same folder as the tester file.

Part B: Saving, with a raise

Background

In Part A, we unrealistically assumed that your salary didn't change. But you are an MIT graduate, and clearly you are going to be worth more to your company over time! So we are going to build on your solution to Part A by factoring in a raise every six months.

In **ps1b.py**, copy your solution to Part A (as we are going to reuse much of that machinery). Modify your program to include the following

1. Have the user input a semi-annual salary raise **semi_annual_raise** (as a decimal percentage)
2. After the 6^{th} month, increase your salary by that percentage. Do the same after the 12^{th} month, the 18^{th} month, and so on.
3. Call the number of months required **months**.

Write a program to calculate how many months it will take you save up enough money for a down payment. Like before, assume that your investments earn a return of **r** = 0.03 (or 3%) and the required down payment percentage is 0.18 (or 18%). Have the user enter the following variables:

1. The starting annual salary (**annual_salary**)
2. The percentage of salary to be saved (**portion_saved**)
3. The cost of your dream home (**total_cost**)
4. The semi-annual salary raise (**semi_annual_raise**)

Notes:

- Retrieve user input.
- Initialize variables with the names given in bold. These variables must be initialized in the following order, at the beginning of your program, before declaring any other variables:
  1. **annual_salary**
  2. **portion_saved**
  3. **total_cost**
  4. **semi_annual_raise**

  You can add additional variables as needed, but they most come after those listed in bold. Be careful about values that represent annual amounts and those that represent monthly amounts.
    - If the variables are not initialized to exactly the names above and as the first few lines, the problem set will be graded incorrectly.
- Be careful about when you increase your salary – this should only happen **after** the 6^{th}, 12^{th}, 18^{th} month, and so on.
- Initialize variables with the names given in bold. These variables must be initialized as the **first few lines of the program**, otherwise the grading software will not work. You can add additional variables as needed.

Test Case 1

```
>>>
Enter your starting annual salary: 110000
Enter the percent of your salary to save, as a decimal: .15
Enter the cost of your dream home: 750000
Enter the semi-annual raise, as a decimal: .03
Number of months: 76
>>>
```

Test Case 2

```
>>>
Enter your starting annual salary: 350000
Enter the percent of your salary to save, as a decimal: .30
Enter the cost of your dream home: 10000000
Enter the semi-annual raise, as a decimal: .05
Number of months: 114
>>>
```

Additionally, open the file ps1_tester.py, and uncomment lines 7 and 8 by erasing the # at the beginning of the line. When you run this file, you should now pass the first six test cases.

Part C: Finding the interest rate you need to succeed

In Part B, you had a chance to explore how both the percentage of your salary that you save each month and your annual raise affect how long it takes you to save for a down payment, given a fixed rate of return on your investments, **r**, (or savings). Now, suppose you want to set a particular goal, e.g. what do I need to be able to afford the down payment in three years? For an **initial_deposit**, how much of a return on your savings will you need to reach your goal of a sufficient down payment in 4 years? In this problem, you are going to write a program to answer that question. To simplify things, assume:

1. The down payment is 0.30 (30%) of the cost of the house
2. The cost of the house that you are saving for is $800K.

You are now going to try to determine the return on savings needed to achieve a down payment on a $800K house in 48 months. Since hitting this exactly is a challenge, we simply want your savings to be within $100 of the required down payment.

In **ps1c.py**, write a program to calculate the lowest return on investment rate. You should use **bisection search** to help you do this efficiently. You should keep track of the number of steps it takes your bisection search to finish. You should be able to reuse some of the code you wrote for part B in this problem.

Because we are searching for a value that is in principle a float, we are going to limit ourselves to two decimals of accuracy (i.e., we may want a return on investment of 7.04% -- or 0.0704 in decimal – but we are not going to worry about the difference between 7.041% and 7.039%). We are also going to assume that our return on investment will lie somewhere between 0% and 100%. Because we only need two decimals of accuracy, this means we can search for an integer between 0 and 10000 (using integer division), and then convert it to a decimal percentage (using float division) when we are calculating the **current_savings** after 4 years. By using this range, there are only a finite number of numbers that we are searching over, as opposed to the infinite number of decimals between 0 and 1. This range will help prevent infinite loops. The reason we use 0 to 10000 is to account for two additional decimal places in the range 0% to 100%. Your code should print out a decimal (e.g. 0.0704 for 7.04%).

Use the following formula for compounded interest in order to calculate the predicted down payment for a given rate of return:

current_savings = **initial_deposit** * (1 + **r**/12)^(number_of_months)

Try different inputs for your **initial_deposit**, and see how the percentage of return on savings, **r**, you need changes in order to reach your desired down payment. Please find the lowest return on investment you need in order to save up enough money for your down payment in three years. Save this value in the variable **best_r**, and the number of steps your bisection search took to find this value in **steps**. Also keep in mind it may not be possible for to save a down payment in three years for some salaries. In this case your function should notify the user that it is not possible to save for the down payment in 4 years with a print statement, and your **best_r** should be equal to the string 'no r possible'. **Please make your program print results in the format shown in the test cases below.**

Note: There are multiple right ways to implement bisection search/number of steps so your results may not perfectly match those of the test case.

Notes

- You can assume that the initial deposit will be always less than down payment + $100.
- There may be multiple rates of return on savings that yield a savings amount that is within $100 of the required down payment on a $1M house. In this case, you can just return any of the possible values.
- Initialize variables with the names given in bold. The following variable must be initialized in the following order, at the beginning of your program, before declaring any other variables:
    - **initial_deposit**
- Retrieve user input.
- Initialize variables with the names given in bold. These variables must be initialized as the **first few lines of the program** otherwise the grading software will not work. You can add additional variables as needed.
- Depending on your stopping condition and how you compute a trial value for bisection search, your number of steps may vary slightly from the example test cases. The grader takes this into account.
- **If a test is taking a long time, you might have an infinite loop! Check your stopping condition.**

Test Case 1

```
>>>
Enter the initial deposit: 40000
Best savings rate: 0.4565 [or very close to this number]
Steps in bisection search: 11 [or very close to this number]
>>>
```

Test Case 2

```
>>>
Enter the initial deposit: 240001
Best savings rate: 0.0
Steps in bisection search: [May vary based on how you implemented your bisection search]
>>>
```

Test Case 3

```
>>>
Enter the initial deposit: 5000
Best savings rate: It is not possible to pay the down payment in four years.
Steps in bisection search: [May vary based on how you implemented your bisection search]
>>>
```

Additionally, open the file ps1_tester.py, and uncomment lines 9 and 10 by erasing the # at the beginning of the line. When you run this file, you should now pass all eight test cases.

Hand-in Procedure

1. Naming Files

Save your solution to part A as **ps1a.py**, to part B as **ps1b.py**, and to part C as **ps1c.py**. **Do not** ignore this step and do not save your files with a different name; otherwise you will receive a 0.

2. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people with whom you collaborated. For example:

```
# Problem Set 1A
# Name: Jane Lee
# Collaborators: John Doe
# Time Spent: 3:30
# Late Days Used: 1 (only if you are using any)
# … your code goes here …
```

3. Submit

Before you upload your files, remove all debugging print statements and other commented out code that is not part of your solution. In addition, open a new console in Spyder and rerun your programs. Be sure to run the tester file as well and make sure the tests all pass. The tester file contains a subset of the tests that will be run to determine the problem set grade.

To submit a file, upload it to the **Problem Sets link on Stellar**. You may upload new versions of each file until the 5 PM deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

To submit a pset with multiple files, you may submit each file individually through the submission page. Be sure to title each submission with the corresponding problem number.

After you submit, please be sure to check the problem set page for the file we have in your submission folder and their contents. When you upload a new file with the same name, your old one will be overwritten.