6.00: **Introduction to Computer Science and Programming**

Problem Set 4

Handed out: Tuesday, September 24, 2008.
Due: 11:59pm, Tuesday, September 30, 2008.

Introduction

This problem set will introduce you to using successive approximation, as well as data structures such as tuples and lists.

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.* For further detail, please review the collaboration policy as stated in the syllabus.

Submission

This problem set, and future ones, will be auto-graded by a test harness. The test harness will expect your files to include just function definitions, with no executable code outside the function definitions. **All your testing code should be in the appropriate test function,** which is provided in the template; for example, the testing code for the futureRetire() problem should be in testFutureRetire(). Be sure to add test cases to these test functions beyond what is provided in the template!

Planning for the future

Recent events in the stock market may seem remote to you, but they underscore the uncertainty of planning for the future. People who had been thinking of retiring in the next year or so may have to rethink those plans, as the value of their 401K accounts drops noticeably. Although retirement may seem a long way off for you, we are going to explore some simple ideas in accruing funds. Along the way, we are going to explore the use of successive approximation methods, as well as the use of some simple data structures, like lists.
We are going to start with a simple model of saving for retirement. Many employers will contribute the equivalent of 5% of your salary to a retirement fund, and then will match any contribution that you add, dollar for dollar, up to an additional 5%. Thus you can salt away up to 15% of your salary into a retirement account (10% of which comes as a bonus from your employer).
We can model a retirement fund with some simple equations. Assume your starting salary is represented by salary; that the percentage of your salary you put into a retirement fund is save; and that the annual growth percentage of the retirement fund is growthRate. Then your retirement fund, represented by the list F, should increase as follows:

	Retirement fund
End of year 1	F[0] = salary * save * 0.01
End of year 2	F[1] = F[0] * (1 + 0.01 * growthRate) + salary * save * 0.01
End of year 3	F[2] = F[1] * (1 + 0.01 * growthRate) + salary * save * 0.01

This process continues each year, with your retirement fund growing both by new contributions and by growth of the principal. Throughout this problem set, growth rates will always be positive (this is not true in the real world, alas!).

Problem 1.

Write a function, called nestEggFixed, which takes four arguments: a salary, a percentage of your salary to save in an investment account, an annual growth percentage for the investment account, and a number of years to work. This function should return a list, whose values are the size of your retirement account at the end of each year, with the most recent year's value at the end of the list.

Complete the implementation of:

def nestEggFixed (salary, save, growthRate, years):

Write your code in the appropriate place in the ps4.py template. To test your function, run the test cases in the test function testNestEggFixed(). You should add additional test cases to this function to further test your code.

This first model is pretty simple. Clearly, the market does not grow at a constant rate. So a better model would be to account for variations in growth percentage each year.

Problem 2.

Write a function, called nestEggVariable, which takes three arguments: a salary (salary), a percentage of your salary to save (save), and a list of annual growth percentages on investments (growthRates). The length of the last argument defines the number of years you plan to work; growthRates[0] is the growth rate of the first year, growthRates[1] is the growth rate of the second year, etc. (Note that because the retirement fund's initial value is 0, growthRates[0] is, in fact, irrelevant.) This function should return a list, whose values are the size of your retirement account at the end of each year.

Complete the implementation of:

def nestEggVariable(salary, save, growthRates):

Write your code in the appropriate place in the ps4.py template. To test your function, run the test cases in the test function testNestEggVariable(). You should add additional test cases to this function to further test your code.

Of course, once you retire you will want to be able to withdraw some amount of money each year for living expenses. You can use your previous code to get the size of your retirement fund when you stop working. Now we want to model how much you can withdraw to spend each year after retirement.

Suppose that, after retirement, your retirement fund continues to grow according to a list of annual growth percentages on investments (growthRates), while your annual expenses are constant (wouldn't zero percent inflation be nice?), called expenses. To see how your retirement fund will change after retirement, we can use the following chart, where we let F represent the size of the retirement fund at the time of retirement, and we let expenses represent the amount of money we withdraw in the first year to cover our living costs:

	Retirement fund
End of year 1	F[0] = savings * (1 + 0.01 * growthRates[0]) – expenses
End of year 2	F[1] = F[0] * (1 + 0.01 * growthRates[1]) – expenses
End of year 3	F[2] = F[1] * (1 + 0.01 * growthRates[2]) – expenses

Problem 3.

Write a function, called postRetirement, which takes three arguments: an initial amount of money in your retirement fund (savings), a list of annual growth percentages on investments while you are retired (growthRates), and your annual expenses (expenses). Assume that the increase in the investment account savings is calculated before subtracting the annual expenditures (as shown in the above table). Your function should return a list of fund sizes after each year of retirement, accounting for annual expenses and the growth of the retirement fund. Like problem 2, the length of the growthRates argument defines the number of years you plan to be retired.

Note that if the retirement fund balance becomes negative, expenditures should continue to be subtracted, and the growth rate comes to represent the interest rate on the debt (i.e. the formulas in the above table still apply).

Complete the definition for:

```
def postRetirement(savings, growthRates, expenses):
```

Write your code in the appropriate place in the ps4.py template. To test your function, run the test cases in the test function testPostRetirement(). You should add additional test cases to this function to further test your code.

Suppose you would like to budget your annual expenses such that by the time you pass on, you will have no retirement funds left (you plan to follow the Andrew Carnegie model and leave no money for your children). One way to determine what your expense budget should be is to try feeding different values of expenses to postRetirement(). You can automate this by using the idea of successive approximation introduced in lecture.

Remember that in lecture, we saw the idea of binary search. In that example, we were trying to find the square root of a number, and we did this by picking a high and low value that we knew bounded the answer (i.e. the answer lay between low and high), and then testing the average of high and low to see how close it was to the right answer (within epsilon absolute difference). If it was close enough, we stopped; if not, we changed the range of values, either making our new low value be the average of the previous low and high, and keeping the old high, or making our new high value the average of the previous low and high, and keeping the old low, depending on the result of our test. You can use the same idea here.

Problem 4.

Write a function, called findMaxExpenses, which takes five arguments: a salary (salary), a percentage of your salary to save (save), a list of annual growth percentages on investments while you are still working (preRetireGrowthRates), a list of annual growth percentages on investments while you are retired (postRetireGrowthRates), and a value for epsilon (epsilon). As with problems 2 and 3, the lengths of preRetireGrowthRates and postRetireGrowthRates determine the number of years you plan to be working and retired, respectively.

Use the idea of binary search to find a value for the amount of expenses you can withdraw each year from your retirement fund, such that at the end of your retirement, the absolute value of the amount remaining in your retirement fund is less than epsilon (note that you can overdraw by a small amount). Start with a range of possible values for your annual expenses between 0 and your savings at the start of your retirement (**HINT #1:** this can be determined by utilizing your solution to problem 2). Your function should print out the current estimate for the amount of expenses on each iteration through the binary search (**HINT #2:** your binary search should make use of your solution to problem 3), and should return the estimate for the amount of expenses to withdraw. (**HINT #3:** the answer should lie between zero and the initial value of the savings + epsilon.) Complete the implementation of:

def findMaxExpenses(salary, save, preRetireGrowthRates, postRetireGrowthRates, epsilon):

Write your code in the appropriate place in the ps4.py template. To test your function, run the test cases in the test function testFindMaxExpenses(). You should add additional test cases to this function to further test your code.

Hand-In Function

1. Save

Save your code in the specific file name indicated for each problem. *Do not ignore this step or save your file(s) with different names.*

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 you collaborated with. For example:

```
# Problem Set 4
# Name: Jane Lee
# Collaborators: John Doe
```

```
    # Time: 1:30
    #
  ... your code goes here ...
```