Homework 4: Loops in Python

See Webcourses and the syllabus for due dates.

General Directions

This homework should be done individually. Note the grading policy on cooperation carefully if you choose to work together (which we don't recommend).
In order to practice for exams, we suggest that you first write out your solution to each problem on paper, and then check it typing that into the computer.
You should take steps to make your Python code clear, including using symbolic names for important constants and using helping functions or procedures to avoid duplicate code.
It is a good idea to test your code yourself first, before running our tests, as it is easier to debug when you run the code yourself.
Tests that are provided in hw4-tests.zip, which consists of several python files with names of the form `test_`*f*`.py`, where *f* is the name of the function you should be writing, and also some other files. Your function *f* should go in a file named *f*`.py` in the same directory as `test_`*f*`.py` and the function itself should be named *f*. These conventions will make it possible to test using pytest.
Pytest is installed already on the Eustis cluster. If you need help installing pytest on your own machine, see the course staff or the running Python page.

Running Pytest from the Command Line

After you have pytest installed, and after you have written your solution for a problem that asks for a function named *f*, you can run pytest on our tests for function *f* by executing at a command line

```
pytest test_f.py > f_tests.txt
```

which puts the output of the testing into the file *f*`_tests.txt`.

Running Pytest from within IDLE

You can also run pytest from within IDLE. To do that first edit a test file with IDLE (so that IDLE is running in the same directory as the directory that contains the files), then from the Run menu select "Run module" (or press the F5 key), and then execute the following statements:

```
import pytest
pytest.main(["test_f.py", "--capture=sys"])
```

which should produce the same output as the command line given above. Then you can copy and past the test output into the file *f*`_tests.txt` to hand in.

What to turn in

For problems that ask you to write a Python procedure, upload your code as an ASCII file with suffix `.py`, and also upload the output of running our tests (as an ASCII file with suffix `.txt`).

Problems

1. [Programming] This question relates for loops, while loops, and recursion in Python, by having you program a function to compute dot products in three different ways.

In Python, we can imagine that lists of numbers represent vectors; for example the list `[0.0, 1.0, 2.0]` represents the vector $\langle 0.0, 1.0, 2.0\rangle$, with 0^{th} element 0.0, 1^{th} element 1.0 and 2^{th} element 2.0. (Recall that Python uses 0-based indexing.)

The dot product function takes two vectors, `v1`, and `v2`, of the same length, and returns the sum `v1[i]` times `v2[i]` for all indexes `i` in **`range`**`(`**`len`**`(v1))`. The argument vectors should be unchanged and the function should not print anything. Examples are shown in Figure 1, where `dpf` is used to stand for a dot product function. This file, and the other testing files mentioned below, is provided in the `hw4-tests.zip` file.

```
# $Id: DotProductTesting.py,v 1.2 2019/02/18 22:10:06 leavens Exp $
from math import isclose
def DotProductTesting(dpf):
    """Test the dot_product function given in the argument dpf.
    Note that dpf will be one of the functions that a student writes."""
    assert dpf([],[]) + 0 == 0 + 0
    assert dpf([1],[1]) == 1
    loi1020 = [10,20]
    loi56 = [5,6]
    assert dpf(loi1020,loi56) == 50+120
    # the arguments are unchanged
    assert loi1020 == [10,20]
    assert loi56 == [5,6]
    assert isclose(dpf([3.14,2.78,9.1],[1.0,2.0,3.0]), 3.14+2.78*2.0+9.1*3.0)
    assert dpf(list(range(5)), list(range(5))) == 0+1+2*2+3*3+4*4
    assert dpf(list(range(10)), list(range(10,20))) \
              == 0+1*11+2*12+3*13+4*14+5*15+6*16+7*17+8*18+9*19
    assert isclose(dpf(list(range(10))+[3.14], list(range(10,20))+[2.781]), \
                   0*10+1*11+2*12+3*13+4*14+5*15+6*16+7*17+8*18+9*19+3.14*2.781)
    assert dpf([-3,-5],[-4,-9]) + 0 == -3*-4 + -5*-9 + 0
```

Figure 1: Tests for a function argument, `dpf`, that is supposed to implement the dot product function.

(a) (10 points) Write the dot product function in Python using a for loop (and without using **`while`** or recursion in the code). Call this function `dot_product_for` and place it in a file named `dot_product_for.py`. Tests for this version are in the file shown in Figure 2.

Your solution must make use of a **`for`** loop in an essential way.

```
# $Id: test_dot_product_for.py,v 1.1 2019/02/18 22:10:06 leavens Exp $
from dot_product_for import *
from DotProductTesting import *
def test_dot_product_for():
    """Testing for dot_product_for."""
    DotProductTesting(dot_product_for)
```

Figure 2: Tests for the function `dot_product_for`, which uses the code in `DotProductTesting.py` shown in Figure 1.

(b) (10 points) Write the dot product function in Python using a while loop (and without using **`for`**, **`range`**, or recursion). Call this function `dot_product_while` and place it in a file named `dot_product_while`. Tests for this version are in the file shown in Figure 3 on the following page.

Your solution must make use of a **while** loop in an essential way.

```
# $Id: test_dot_product_while.py,v 1.1 2019/02/18 22:10:06 leavens Exp $
from dot_product_while import *
from DotProductTesting import *
def test_dot_product_while():
    """Testing for dot_product_while."""
    DotProductTesting(dot_product_while)
```

Figure 3: Tests for the function `dot_product_while`, which use the code in `DotProductTesting.py` shown in Figure 1 on the preceding page.

(c) (10 points) Write the dot product function in Python using recursion (and without using a loop of any kind). Call this function `dot_product_rec` and place it in a file named `dot_product_rec`. Tests for this version are in the file shown in Figure 4.

Your solution must make use of recursion in an essential way.

```
# $Id: test_dot_product_rec.py,v 1.2 2019/02/19 05:23:38 leavens Exp $
from dot_product_rec import *
from DotProductTesting import *
def test_dot_product_rec():
    """Testing for dot_product_rec."""
    DotProductTesting(dot_product_rec)
```

Figure 4: Tests for the function `dot_product_rec`, which use the code in `DotProductTesting.py` shown in Figure 1 on the previous page.

If you find it useful, you can use the helping functions shown in Figure 5 on the following page, which define the functions `isEmpty`, `first`, and `tail` for Python lists. Note, however, that these are not methods, so you would write `isEmpty(v)` to test if a vector `v` is empty. This file is included in the `hw4-tests.zip` file.

Remember to turn in all the source code files: `dot_product_for.py`, `dot_product_while.py`, and `dot_product_rec.py` as well as the output of running each of the tests with `pytest`. Your code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

2. (20 points; extra credit) (extra credit) The code for a recursive function `f` is said to be *tail-recursive* if whenever it calls itself, it has no more computation to do after the recursive call returns. Thus, if each of the calls of `f` from within the code for `f` are of the form **return** `f(...)`, then the function `f` is tail-recursive.[1] Another way to say this is that recursive calls to `f` need not send their results to any "pending computation," since they are immediately used in a return. Tail recursion can often be accomplished by using a helping function that is tail recursive and has extra arguments. These extra arguments correspond to the local variables (often accumulators and counters) used in writing a while loop.

   For this extra credit problem, write and test, using the file `test_dot_product_tr`, which is shown in Figure 6 on the next page, a tail-recursive implementation of the dot product function, named `dot_product_tr`. (Note, the code for this function must not use any loops, but only tail-recursion.)

   Turn in both your source code and the results of running our tests.

[1] Some functional language compilers and interpreters optimize tail-recursive calls.

```
# $Id: seq_ops.py,v 1.2 2019/02/18 19:51:21 leavens Exp $
# Some helping functions for Python sequences
# that make them look more like LispLists.
# These all work on sequences of any element type.

def isEmpty(seq):
    """type: sequence(T) -> bool
    Ensures: result is len(seq) > 0"""
    return len(seq) == 0

def first(seq):
    """type: sequence(T) -> T
    Requires: len(seq) > 0
    Ensures: result is seq[0]"""
    return seq[0]

def tail(seq):
    """type: sequence(T) -> sequence(T)
    Requires: len(seq) > 0
    Ensures: result is seq[1:]"""
    return seq[1:]
```

Figure 5: Helping functions on Python sequences that implement functions similar to the methods on LispLists.

```
# $Id: test_dot_product_tr.py,v 1.1 2019/02/18 22:10:06 leavens Exp $
# Note that this is an EXTRA CREDIT PROBLEM, you don't have to do it!
from dot_product_tr import *
from DotProductTesting import *
def test_dot_product_rec():
    """Testing for dot_product_tr."""
    DotProductTesting(dot_product_tr)
```

Figure 6: Tests for the function `dot_product_tr`, which use the code in `DotProductTesting.py` shown in Figure 1 on page 2.

3. (15 points) [Programming] The *graph of a function f for a list of arguments* a is a list of pairs of the form $(a_i, f(a_i))$, where a_i is the i^{th} element of a.

   Your task in this problem is to write a Python function, `graph(f, argList)`, of

   type: `(T->U, list(T)) -> list(pair(T,U))`

   for all types `T` and `U`, that takes a function, `f`, and a list of arguments, `argList`, and returns a list of pairs, such that for all `i` in **`range`**`(0,`**`len`**`(argList))`, the `i`th element of the result is the pair `(argList[i], f(argList[i]))`. Tests for `graph` are given in Figure 7 on the following page.

   Hints: Accumulate a list of the pairs in a variable.

4. (25 points) [Programming] The derivative of a function `f` at a point `x` can be (numerically) approximated by the function `easydiff` shown in Figure 8 on page 6. Good approximations are given by small values of `delta`, but if `delta` is too small, then rounding errors may swamp the result.

```
# $Id: test_graph.py,v 1.1 2019/02/19 03:14:23 leavens Exp $
from graph import graph
def test_graph():
    """Testing for graph."""
    # the functions used in this test are defined below...
    assert graph(plus1,[]) == []
    assert graph(plus1,[5, 10, 11, 9, 10]) \
        == [(5,6), (10,11), (11,12), (9,10), (10,11)]
    loi37420 = [3, 7, 4, 2, 0]
    assert graph(odd, loi37420) \
        == [(3,True), (7,True), (4,False), (2,False), (0,False)]
    # the argument is unchanged
    assert loi37420 == [3, 7, 4, 2, 0]
    assert graph(multBy3, list(range(6))) \
        == [(0,0), (1,3), (2,6), (3,9), (4,12), (5,15)]
    loi323000 = list(range(32,3000))
    gmb3big = graph(tnp1, loi323000)
    for i in loi323000:
        n = i-32
        assert gmb3big[n] == (i, tnp1(i))
    loi1100 = list(range(1,100))
    ghail = graph(hailstone, loi1100)
    for j in loi1100:
        m = j-1
        assert ghail[m] == (j, hailstone(j))

# some helping functions for testing
def plus1(n):
    return n+1

def multBy3(n):
    return n*3

def odd(n):
    return n % 2 == 1

def tnp1(n):
    if odd(n):
        return plus1(multBy3(n))
    else:
        return n

def hailstone(n):
    return tnp1(n) // 2
```

Figure 7: Tests for `graph`.

One way to choose `delta` is to compute a sequence of approximations, starting with a reasonably large one, and keep halving `delta` until the result stabilizes.

Your task in this problem is to write a Python function `approxDerivAt(delta, f, x, epsilon)` of **`type`**`: (`**`float`**`, `**`float`**` -> `**`float`**`, `**`float`**`, `**`float`**`) -> `**`float`** that takes an initial value of `delta`, which is a float that is strictly greater than 0.0, a function `f` that takes a float and returns a float, a float `x`, and a value `epsilon` that is strictly greater than 0.0 and returns an approximation to the derivative of `f` at `x` that is accurate to within `epsilon`.

```
# $Id: easydiff.py,v 1.1 2019/02/19 04:39:32 leavens Exp $
def easydiff(f, x, delta):
    """type: (float -> float, float, float) -> float
    Requires: delta > 0.0
    Ensures: result is (f(x+delta) - f(x)) / delta"""
    return (f(x+delta) - f(x)) / delta
```

Figure 8: The `easydiff` function. This function is provided in the file `hw4-tests.zip`.

Hint: use a **`while`** loop and the `fabs` function from the `math` module to compute absolute values of differences. Keep halving delta while testing the result of `easydiff(f, x, delta)` until the value stabilizes in the sense that:

```
fabs(easydiff(f,x,delta) - easydiff(f,x,delta/2)) < epsilon
```

is True. Then return the value of `easydiff(f,x,delta/2)`.

Tests for `approxDerivAt` are shown in Figure 9 on the following page.

Points

This homework's total points: 70.

References

[Hay84] B. Hayes. Computer recreations: On the ups and downs of hailstone numbers. *Scientific American*, 250(1):10–16, January 1984.

[LV92] Gary T. Leavens and Mike Vermeulen. $3x + 1$ search programs. *Computers and Mathematics with Applications*, 24(11):79–99, December 1992.

```
# $Id: test_approxDerivAt.py,v 1.1 2019/02/19 04:39:32 leavens Exp $
from math import isclose
from approxDerivAt import *

def test_approxDerivAt():
    """Testing for approxDerivAt."""
    eps = 1.0e-5
    # The functions used as the second argument are found below...
    assert isclose(approxDerivAt(1.0, id, 2.0, eps), 1.0, abs_tol=eps)
    assert isclose(approxDerivAt(1.0, id, 0.0, eps), 1.0, abs_tol=eps)
    assert isclose(approxDerivAt(4.0, linear, 0.0, eps), 1.0, abs_tol=eps)
    assert isclose(approxDerivAt(1.0, square, 3.0, eps), 6.0, abs_tol=eps)
    assert isclose(approxDerivAt(1.0, square, 5.0, eps), 10.0, abs_tol=eps)
    assert isclose(approxDerivAt(1.0, square, 0.25, eps), 0.5, abs_tol=eps)
    assert isclose(approxDerivAt(1.0, xp1TimesXm1, 5.0, eps), 10.0, abs_tol=eps)
    assert isclose(approxDerivAt(4.0, pow4, 10.0, eps), 20.0*10.0**3+6.0*10.0**2, abs_tol=eps)
    assert isclose(approxDerivAt(4.0, pow4, 2.0, eps), 20.0*2.0**3+6.0*2.0**2, abs_tol=eps)
    assert isclose(approxDerivAt(4.0, pow4, 2.0, 0.1), 20.0*2.0**3+6.0*2.0**2, abs_tol=0.1)
    assert isclose(approxDerivAt(1.0, pow4, 0.5, 0.1), 20.0*0.5**3+6.0*0.5**2, abs_tol=0.1)

# functions of type float -> float for use in testing
def id(x):
    return x

def linear(x):
    return x+2

def square(x):
    return x*x

def xp1TimesXm1(x):
    return x*x - 1

def pow4(x):
    return 5.0*x**4.0 + 2.0*x**3.0
```

Figure 9: Tests for approxDerivAt.