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` 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 while and for loops in Python, by having you program the factorial function two different ways. The factorial function takes an integer, n, and returns the product $1 \times 2 \times 3 \times \cdots \times n-1 \times n$ (which is often written as $n!$ in mathematics and statistics). It is assumed that $n \geq 1$. Examples are shown in Figure 1 on the following page.

```
# $Id: factorialTesting.py,v 1.1 2017/02/21 03:48:09 leavens Exp leavens $
def factorialTesting(fact):
    """Test the factorial function given in the argument fact."""
    assert fact(1) == 1
    assert fact(2) == 2
    assert fact(3) == 6
    assert fact(4) == 24
    assert fact(5) == 120
    assert fact(6) == 720
    assert fact(7) == 720*7
    assert fact(8) == 720*7*8
    assert fact(10) == 1*2*3*4*5*6*7*8*9*10
    assert fact(20) == fact(10)*11*12*13*14*15*16*17*18*19*20
    assert fact(27) == 10888869450418352160768000000
    assert fact(32) == 263130836933693530167218012160000000
```

Figure 1: Tests for a function argument, `fact`, that is supposed to implement the factorial function.

(a) (10 points) Write the factorial function in Python using a while loop (and without using **`for`** or **`range`**, except that **`range`** may be used in assertions). Call this function `factorial_while` and place it in a file named `factorial_while.py`. Tests for this version are in the file shown in Figure 2.

```
# $Id: test_factorial_while.py,v 1.1 2017/02/21 03:48:09 leavens Exp leavens $
from factorial_while import *
from factorialTesting import *
def test_factorial_while():
    """Testing for factorial_while."""
    factorialTesting(factorial_while)
```

Figure 2: Tests for the function `factorial_while`, which use the code in `factorialTesting.py` shown in Figure 1.

(b) (10 points) Write the factorial function in Python using a for loop (and without using **`while`**). Call this function `factorial_for` and place it in a file named `factorial_for.py`. Tests for this version are in the file shown in Figure 3 on the following page.

Hint: you may also find **`range`** helpful for this part, but your solution must make use of a **`for`** loop in an essential way.

Remember to turn in both source code files `factorial_while.py` and `factorial_for.py`, and the output of running our 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) 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 is tail-recursive.[1] 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.

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

```
# $Id: test_factorial_for.py,v 1.1 2017/02/21 03:48:09 leavens Exp leavens $
from factorial_for import *
from factorialTesting import *
def test_factorial_for():
    """Testing for factorial_for."""
    factorialTesting(factorial_for)
```

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

For this extra credit problem, write and test, using the file `test_factorial_rec`, a tail-recursive implementation of the factorial function, named `factorial_rec`. (Note, the code for this function should not use loops, but only tail-recursion.)

```
# $Id: test_factorial_rec.py,v 1.1 2017/02/21 03:48:09 leavens Exp leavens $
# Note that this is an EXTRA CREDIT PROBLEM, you don't have to do it!
from factorial_rec import *
from factorialTesting import *
def test_factorial_rec():
    """Testing for factorial_rec."""
    factorialTesting(factorial_rec)
```

Figure 4: Tests for the function `factorial_rec`, which use the code in `factorialTesting.py` shown in Figure 1 on the preceding page.

3. (15 points) [Programming] The hailstone, or $3x + 1$ function, is the function shown in Figure 5. (Note

```
# $Id: hailstone.py,v 1.1 2017/02/21 03:03:39 leavens Exp $
def hailstone(x):
    """type: int -> int
    Requires 0 < x
    Ensures: if x is odd, then result is (3*x+1)//2, otherwise result is x/2."""
    return (3*x+1)//2 if x % 2 == 1 else x // 2
```

Figure 5: The hailstone or $3x + 1$ function.

that if `x` is odd, then `3x+1` is even, and so can be divided by 2 to yield an integer, as can each even number.) This function was described in a 1984 *Scientific American* article by Hayes [Hay84] as puzzling mathematicians for decades. If you iterate this function, forming the *trajectory* consisting of the numbers `n`, `hailstone(n)`, `hailstone(hailstone(n))`, etc., then the numbers reached seem to go up and down, like hailstones in a thunderstorm, until they come crashing back to 1 (the earth).[2] For example, the trajectory of 3 is [3, 5, 8, 4, 2, 1] because `hailstone(3)` is 5, and `hailstone(8)` is 4, and `hailstone(4)` is 2, etc.

Your task in this problem is to write a Python function, `trajectory(n)`, of

[2] It is unknown whether they do always reach 1 in this process, but this has been experimentally verified for numbers up to a trillion [LV92].

type: **int** -> **list**(**int**)

that takes an int value, n, and returns the list

`[n, hailstone(n), hailstone(hailstone(n)), ..., 1]`

(provided that iterating the hailstone function in this way leads back to 1 when starting from n). Tests for `trajectory` are given in Figure 6.

```
# $Id: test_trajectory.py,v 1.1 2017/02/21 03:03:39 leavens Exp $
from trajectory import trajectory
def test_trajectory():
    """Testing for trajectory."""
    assert trajectory(1) == [1]
    assert trajectory(2) == [2,1]
    assert trajectory(3) == [3, 5, 8, 4, 2, 1]
    assert trajectory(5) == [5, 8, 4, 2, 1]
    assert trajectory(7) == [7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1]
    assert trajectory(8) == [8, 4, 2, 1]
    assert trajectory(16) == [16, 8, 4, 2, 1]
    assert trajectory(17) == [17, 26, 13, 20, 10, 5, 8, 4, 2, 1]
    assert trajectory(26) == [26, 13, 20, 10, 5, 8, 4, 2, 1]
    assert trajectory(27) == [27, 41, 62, 31, 47, 71, 107, 161, 242, \
                              121, 182, 91, 137, 206, 103, 155, 233, 350, \
                              175, 263, 395, 593, 890, 445, 668, 334, 167, \
                              251, 377, 566, 283, 425, 638, 319, 479, 719, \
                              1079, 1619, 2429, 3644, 1822, 911, 1367, 2051, \
                              3077, 4616, 2308, 1154, 577, 866, 433, 650, \
                              325, 488, 244, 122, 61, 92, 46, 23, 35, 53, \
                              80, 40, 20, 10, 5, 8, 4, 2, 1]
    assert len(trajectory(703)) == 109
    assert len(trajectory(871)) == 114
    assert len(trajectory(26623)) == 195
```

Figure 6: Tests for `trajectory`.

Hints: Use a **while** loop and accumulate a list. Your code should import and call the `hailstone` function in Figure 5 on the previous page. The file `hailstone.py` is provided in the `hw4-tests.zip` file.

4. (25 points) [Programming] Define a Python function, `ispalindrome(st)`, of

   type: `string` -> **bool**

   which takes a string, `st`, and returns True just when `st` is a palindrome, such as "madam", "nurses run", or "Was it a car or a cat I saw?". These strings read the same backwards and forwards, when spaces and punctuation are removed and differences in capitalization are ignored.

   Hint: we suggest that your code first makes a copy of `st` that has no whitespace or punctuation characters. The strings `string.whitespace` and `string.punctuation` in the built-in Python module `string`. (See https://docs.python.org/3/library/string.html for details.) You may find one of the Python membership operators **in** or **not in** helpful. You can also use the method `.casefold()` on a string to change the string into a form suitable for case-insensitive testing (since it is a method, you would write `st.casefold()` to return a case-folded copy of the string `st`). It will be helpful to write one or two helping functions to accomplish these tasks. Note that we will take points off if your code is unclear and using helping functions can help clarify what you are doing.

   Tests for `ispalindrome` are shown in Figure 7 on the following page.

```
# $Id: test_ispalindrome.py,v 1.1 2017/02/21 02:01:50 leavens Exp $
from ispalindrome import ispalindrome
def test_ispalindrome():
    """Tests for ispalindrome."""
    assert ispalindrome("")
    assert ispalindrome("a")
    assert ispalindrome("aa")
    assert ispalindrome("a, a!")
    assert ispalindrome("aba")
    assert not ispalindrome("aces")
    assert ispalindrome("dood")
    assert ispalindrome("madam")
    assert ispalindrome("Madam, I'm Adam.")
    assert ispalindrome("A man, a plan, a canal -- Panama!")
    assert not ispalindrome("A man, some plan, a canal -- Panama!")
    assert not ispalindrome("Fetch my thermos; it's hot out there.")

def test_wikipalindromes():
    """Tests from Wikipedia's article on palindromes."""
    assert ispalindrome("Was it a car or a cat I saw?")
    assert ispalindrome("No 'x' in Nixon")
    assert ispalindrome("Sator Arepo Tenet Opera Rotas")
    assert not ispalindrome("The sower Arepo holds with effort the wheels")
    assert ispalindrome("In girum imus nocte et consumimur igni")
    assert not ispalindrome("we go wandering at night and are consumed by fire")
    assert ispalindrome("Mr. Owl ate my metal worm!")
    assert ispalindrome("Go hang a salami, I'm a lasagna hog.")
    assert ispalindrome("Able was I, ere I saw Elba...")
```

Figure 7: Tests for `ispalindrome`.

Points

This homework's total points: 80.

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.