Python Programming

Homework 5

Instructor: Dr. Ionut Cardei

The following problems are from Chapters 13 (Programming with Classes) and 14 (Files and Exceptions II).

For full credit:

- make sure you follow exactly all requirements listed in this document,
- the program must have all required features and run as expected.

Write your answers into a new Word document called **h5.doc**.
Write your full name at the top of the file and add a heading before each problem solution.

For each problem:

- write a **subtitle** with the problem number
- insert the code from p1.py and p2.py
- insert any additional items, such as screenshots

Convert the doc file to PDF.
For this homework you need to upload on Canvas the PDF file and the Python .py source files (p1.py, p2.py,…).

Don't forget to read the **Important Notes** at the end of this file.

Problem 1. Prey-Predators-Humans Problem

Take the prey-predator problem described in Chapter 13 and add humans. Use the file program13-14.py, attached to the assignment page.
A human is a predator that eats prey. It also moves, breeds like an Animal, and kills predators (e.g. to keep prey for humans or for fun).
Here are the detailed rules pertaining to humans:

1. A Human object moves like an Animal object on the island grid.
2. A Human object eats Prey just like a Predator object, with starving time given by a Human.starve_time class attribute, initialized in main().
3. A Human object breeds like an Animal object, with the breeding period given by a Human.breed_time class attribute, initialized in main().
4. A Human object hunts Predator objects periodically. Every Human.hunt_time clock ticks (starting with the Human object's creation time), if a Predator is in a neighboring cell (using check_grid()),

the Human will move to its cell and remove the Predator from the island, in the same way Predators eat Prey objects.

*** All other rules for Prey and Predators remain in effect. ***

We want to study the impact of humans on the island animal populations. Add the following to the Island class:

- Proper initialization for Human objects. The constructor and *init_animals*() should take each an extra parameter *count_humans* and *init_animals*() should position Human objects at random positions.
- A method *count_humans*() that returns the number of Human objects on the grid.

The *main*() method should:

- Take extra parameters for the Human class attributes described above and should properly initialize them.
- Keep track of the Human population for each clock tick in the same ways it's done for Predator and Prey objects.
- NOT stop the simulation if Human objects or Predator get extinct, but only when Prey dissapear.
- Display at the end with pylab the evolution in time of the Prey, Predator, and Human populations.
- Display the island grid to the terminal, at the beginning and at the end of the simulation.

More requirements:

1. Add a Human class to the existing class hierarchy so that code is properly reused. Use the problem description above to decide what class is Human's superclass.
2. Use the object-oriented design process. Integrate class Human smoothly into the existing design and code.
3. No copy-pasted code from other classes is allowed in class Human.
4. Print a Human object on the Island grid with character 'H'.
5. Apply the proper coding style and techniques taught in this class.
6. Write docstrings for functions and comment your code following the guidelines from the textbook.

Preparation Instructions:

1. Write your code in file p1.py.
2. Insert p1.py into h5.doc.
3. Run the program with different combinations of parameters and find an interesting case.
4. Take a screenshot with the pylab chart showing the populations vs. time. Insert this screenshot in h5.doc.
5. Take a screenshot with the terminal showing the island grid printout (first tick and last tick) + any other statistics displayed in main(). Insert this screenshot in h5.doc.

Problem 2. Non-interactive Text Editing

A non-interactive text editor executes simple editing commands on a text file using a programming interface (i.e. functions or object methods) in some programming language, like Python. All commands take as input, at minimum, the file name as a parameter, then perform their job (reading or modifying the file), and end returning some results. For this problem we assume the text files have the utf-8 encoding (the default in Python) and they are not too large. Therefore, it's OK to read the file entirely to memory, execute the operations required, then overwrite the file with the modified file content, as a string. This is not the only approach, but it's the simpler one and it works for smaller files.

Here are the editing commands, with the corresponding function signatures:

- *ed_read*(*filename*, *from*=0, *to*=-1): returns as a string the content of the file named *filename,* with file positions in the half-open range [*from*, *to*). If *to* == -1, the content between *from* and the end of the file will be returned. If parameter *to* exceeds the file length, then the function raises exception *ValueError* with a corresponding error message.
- *ed_find*(*filename*, *search_str*): finds string *search_str* in the file named by *filename* and returns a list with index positions in the file text where the string *search_str* is located. E.g. it returns [4, 100] if the string was found at positions 4 and 100. It returns [] if the string was not found.
- *ed_replace*(*filename*, *search_str, replace_with, occurrence=-1)*: replaces *search_str* in the file named by *filename* with string *replace_with*. If *occurrence==-1,* then it replaces ALL occurrences. If *occurrence>=0,* then it replaces only the occurrence with index *occurrence*, where 0 means the first, 1 means the second, etc. If the *occurrence* argument exceeds the actual occurrence index in the file of that string, the function does not do the replacement. The function returns the number of times the string was replaced.
- *ed_append*(*filename*, *string*): appends *string* to the end of the file. If the file does not exist, a new file is created with the given file name. The function returns the number of characters written to the file.
- *ed_write*(*filename*, *pos_str_col*): for each tuple (*position*, *s*) in collection *pos_str_col* (e.g. a list) this function writes to the file at position *pos* the string *s*. This function overwrites some of the existing file content. If any *position* parameter is invalid (< 0) or greater than the file contents size, the function does not change the file and raises *ValueError* with a proper error message. In case of no errors, the function returns the number of strings written to the file. Assume the strings to be written do not overlap.
- *ed_insert*(*filename*, *pos_str_col*): for each tuple (*position*, *s*) in collection *pos_str_col* (e.g. a list) this function *inserts* into to the file content the string *s* at position *pos*. This function does **not** overwrite the existing file content, as seen in the examples below. If any *position* parameters is invalid (< 0) or greater than the original file content length, the function does not change the file at all and raises *ValueError* with a proper error message. In case of no errors, the function returns the number of strings inserted to the file.

For all the functions above, the file should not be changed in case of an error. If an error related to file I/O (e.g. *FileNotFoundError* or *IOError*) occurs in one of these functions, it should be passed to the caller. This means that these functions should not catch (except:) file IO errors.

Examples:

```
fn = "file1.txt"    # assume this file does not exist yet.
ed_append(fn, "0123456789")    # this will create a new file
ed_append(fn, "0123456789")    # the file content is: 01234567890123456789

print(ed_read(fn, 3, 9))    # prints 345678. Notice that the interval excludes index to (9)
print(ed_read(fn, 3))       # prints from 3 to the end of the file: 34567890123456789

lst = ed_find(fn, "345")
print(lst)                  # prints [3, 13]
print(ed_find(fn, "356"))   # prints []

ed_replace(fn, "345", "ABCDE", 1)  # changes the file to 0123456789012ABCDE6789

# assume we reset the file content to 01234567890123456789  (not shown)
ed_replace(fn, "345", "ABCDE")  # changes the file to 012ABCDE6789012ABCDE6789

# assume we reset the file content to 01234567890123456789 (not shown)
# this function overwrites original content:
ed_write(fn, ((2, "ABC"), (10, "DEFG")))   # changes file to: 01ABC56789DEFG456789
# this should work with lists as well: [(2, "ABC"), (10, "DEFG")]

# assume we reset the file content to 01234567890123456789 (not shown)
ed_write(fn, ((2, "ABC"), (30, "DEFG")))  # fails. raises ValueError("invalid position 30")

# assume we reset the file content to 01234567890123456789 (not shown)
# this function inserts new text, without overwriting:
ed_insert(fn, ((2, "ABC"), (10, "DEFG")))
# changed file to: 01ABC23456789DEFG0123456789
```

Here are your tasks for this problem:

a) Implement in Python all functions listed on the previous page in file p2.py. Write the contracts for each function in its **docstring**. Ensure precondition errors are handled properly and ValueError is raised in case the precondition is not met. All errors related to file I/O (e.g. *FileNotFoundError* or *IOError*) should be passed to the caller.

b) Write test functions for functions ***ed_write*** and ***ed_replace*** using the ***testif***() function we used for homework 3, listed in Appendix A. These test functions should be named *test_x* , where x is the name of the function tested. E.g. *test_ed_write*() should use ***testif***() to test *ed_write*(). In general, this type of test functions are called unit tests as they test just one function (or one method in a class). In Appendix B there is a sample unit test for *ed_append*(). Use that as a template.

c) Write a main() function where you show how to use all *ed_….* functions written for part a).

d) EXTRA CREDIT for 5 points

Write a function *ed_search*(*path*, *search_string*) that searches for *search_string* in all files in directory *path* and its immediate subdirectories using the *os.walk* function.

The *ed_search* function must use the *ed_find*() function from part a). *ed_search* must return the list of full (absolute) path names where files in which *search_string* is found or the empty list [] if the string is not found in any files. The *ed_search* function should **not** proceed recursively to all subdirectories.

Submission Instructions

Convert the **h5.doc** file to PDF format (file **h5.pdf**) and upload the following files on Canvas by clicking on the Homework 5 link:

1. h5.pdf
2. p1.py
3. p2.py

Grading (100 points max + extra credit):

Problem 1: 50%

- code correctness: 35%
- coding style and following standards: 7%
- screenshots: 8%

Problem 2: 50% + 5 extra credit points

- code correctness: 40%
- coding style and following standards: 10%

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Upload the PDF, **and .py source files** on Canvas.
- Only submissions uploaded before the deadline will be graded.

- You have unlimited attempts to upload this assignment, but only the last one uploaded before the deadline will be graded.

APPENDIX A

The *testif* function used for writing unit tests:

```
def testif(b, testname, msgOK="", msgFailed=""):
    """Function used for testing.
    param b: boolean, normally a tested condition: true if test passed, false otherwise
    param testname: the test name
    param msgOK: string to be printed if param b==True  ( test condition true )
    param msgFailed: string to be printed if param b==False   ( test condition false )
    returns b
    """
    if b:
        print("Success: "+ testname + "; " + msgOK)
    else:
        print("Failed: "+ testname + "; " + msgFailed)
    return b
```

APPENDIX B

Sample unit test for Problem 2, testing the *ed_append*() function:

```
# unit test for append.
# This is superficial - it does not deal with file I/O errors.
def test_append():
    test_name = "test_append"
    test_fn = "test.txt"     # test file name
    if os.path.isfile(test_fn):
        os.remove(test_fn)            # start from scratch each test, with a brand new file.

    # write some initial content to the file
    initial_text = "ABCD"
    with open(test_fn, "w") as test_f:
        test_f.write(initial_text)

    # now test ed_append:
    try:
        new_text = "01234"
        # our test subject:
        ret = ed_append(test_fn, new_text)

        expected_text = initial_text + new_text

        # we need to check returned value and that the file has changed accordingly:
        current_text = open(test_fn, "r").read()   # read entire file

        # test the condition:
        cond = (ret == len(new_text)) and (current_text == expected_text)

        return testif(cond, test_name)
    except Exception as exc:           # catch all
        print("test {} failed due to exception: {}\n".format(test_name, str(exc)))
        return False
```