Fundamentals of Computer Science - programming exam

Instructions

IMPORTANT - BEFORE YOU START:

1. Open and edit the `.py` file in Spyder to solve the exercises.
2. Upload the `.py` file to BBoard.

Additional notes

- The exam is **not open book**! No notes, smartphones, pen drives, ...
- **Clean your mess** after you finish! Comments and tests/asserts are welcome, but unnecessary code should be deleted.
- **Tests are provided** for each exercise, although they are initially commented out. They are very helpful to understand the intended behavior and to check the correctness of your implementation. Make sure to uncomment and run them as you go though the exercises..

Part 1 (40min)

Note: For the exercises in part 1, do not make use of libraries, python's built-in functions, sets, dictionaries, list's methods, etc.. unless otherwise stated. Just plain loops and if/else statements.

Ex. 1.1 - Median

Write a function called `median` that takes a list of integers as its only argument and returns the median of the list.

If the list contains non-integer elements raise an exception.

In order to compute the median, the list has to be sorted (use the `sorted` function) and the midpoint returned. If the list has an even number of elements, return the average of the middle two elements.

Examples of the desired behavior:

```
In[1]: l = [1, 5, 7, 2, 9]
In[2]: median(l)
Out[2]: 5

In[3]: l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
In[4]: median(l)
Out[4]: 5.5

In[4]: median(1, 2, 2.1) # displays an error
```

Execise 1.2 Atbash Cypher

IMPORTANT: for this exercise, you must use a couple of functions that allow you to convert between a string character and an integer. These functions are the built-in function `ord` (takes a character, returns an integer) and `chr` (takes an integer, returns a character). When using lowercase letters, the integers returned by `ord` are all in sequence.

Examples:

```
In [1]: ord('a')
Out[1]: 97
In [2]: ord('b') # SAME AS ord('a')+1
Out[2]: 98
In [3]: ord('c') # SAME AS ord('a')+2
Out[3]: 99
In [4]: ord('z') # SAME AS ord('a')+25
Out[4]: 122
In [5]: chr(97) # INVERSE OF ord('a')
Out[5]: 'a'
In [6]: chr(98) # INVERSE OF ord('b')
Out[6]: 'b'
In [7]: chr(122) # INVERSE OF ord('z')
Out[7]: 'z'
```

Your task is to write a function called `atbash` that takes a string as input, and returns another string which is encoded via the “Atbash cipher”, in which each letter is mapped to its reverse in terms of alphabetical order:

```
'a' → 'z'
'b' → 'y'
'c' → 'x'
⋮
'x' → 'c'
'y' → 'b'
'z' → 'a'
```

The function should make sure that the input argument is a string, and give an error otherwise. You can assume that all inputs are lowercase letters of the English alphabet, you don’t need to check that. You cannot use dictionaries or lists that contain the whole mapping, you need to compute it with the help of `ord` and `chr`.

Tip: The tricky part here is computing the mapping. Work out the solution for a single character at first.

Examples of the desired behavior:

```
In [1]: atbash(123) # THIS MUST GIVE AN ERROR (WRONG ARGUMENT)

In [2]: atbash('abc')
Out[2]: 'zyx'
```

```
In [3]: atbash('abba')
Out[3]: 'zyyz'

In [4]: atbash('atbash')
Out[4]: 'zgyzhs'

In [5]: atbash('abcdefghijklmnopqrstuvwxyz')
Out[5]: 'zyxwvutsrqponmlkjihgfedcba'

In [6]: atbash(atbash('twice')) # APPLYING IT TWICE RETURNS THE ORIGINAL STRING
Out[6]: 'twice'
```

Exercise 1.3 Maximum sum sublist

Write a function `max_sum_sublist` that takes a list of integers and an integer `k`, and returns the maximal sum of any contiguous sublist of length `k`.

The solution should involve a single loop (no nested loops).

Remember that you cannot use list slicing and call functions like `sum`.

```
assert max_sum_sublist([1, 2, 3, 4, 5], 2) == 9
assert max_sum_sublist([1, 2, 3, 4, 5], 3) == 12
assert max_sum_sublist([11, 5, 12, 4, 5], 2) == 17
```

Exercise 1.4 - Magic Square

Given a square matrix of size `n x n`, represented as a list of lists, write a function `ismagic` to check if it is a magic square (i.e., the sum of each row, column, and both diagonals is the same).

The function should return `True` if the matrix is a magic square, and `False` otherwise.

```
assert ismagic([[2, 7, 6], [9, 5, 1], [4, 3, 8]]) == True
assert ismagic([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) == False
```

Part 2 (50 min)

Part 2 Exercise 1 - Complex

We define a class `Complex` representing a complex number (you don't have to know what a complex number is). Below is a list of what you have to do. No need to the check the type of the methods' inputs.

1. Define the constructor of the class to take two arguments (some int or float numbers) that we call `re` and `im` (these names stand for the real and imaginary part respectively). Store them as internal fields.
2. Implement the appropriate special method so that a complex number is printed as follows in the console:

```
In [1]: Complex(2, 3)
Out[1]: 2 + 3i

In [2]: Complex(3.7, 4.8)
Out[2]: 3.7 + 4.8i
```

3. Implement the comparison operator `==` among two complex numbers. Two complex numbers are equal if their `re` fields and their `im` fields are equal.
4. Implement the `*` operator among two complex numbers. Let's call `re1`, `re2` , `im1` and `im2` the `re` fields and the `im` field of the two complex being added. The result of the multiplication should be a complex number with `re = re1*re2 - im1*im2` and `im = re1*im2 + im1*re2`.

Exercise 2.2 - Lousy File System

The task in this exercise is to create a `LousyFS` class that implements a “file system”, and an auxiliary `Directory` class. (A file system is a way to organize data into files and directories inside a disk.)

1. Write a class called `Directory`. Its constructor takes no arguments, and simply creates an empty `dict`, called `items`, inside the object.

```
In [1]: d = Directory()
In [2]: d.items
Out[2]: {}
```

This `dict` will be used to store files or other directories. Its keys will be strings (the files/directories names) and the values will either be strings (representing a file’s contents) or other `Directory` objects.

2. Write two methods of the `Directory` class, called `additem` and `delitem`.

`additem` takes two arguments, `name` (a string) and `contents` (either a string or another `Directory` object). It should check that none of the keys in the internal `items` dict is equal to `name` (giving an error otherwise). Then is should add the contents to the internal dict, with the key `name`.

`delitem` takes only a `name` argument and removes the corresponding item from the internal dict.

```
In [3]: d.additem("my_password.txt", "12345") # ADDS A FILE

In [4]: d.additem("docs", Directory()) # ADDS A DIR

In [9]: d.items.keys()
Out[9]: dict_keys(['my_password.txt', 'docs'])

In [10]: d.additem("docs", "abcd") # MUST GIVE AN ERROR, THE NAME docs IS ALREADY
TAKEN

In [11]: d.delitem("docs")

In [12]: d.items
Out[12]: {'my_password.txt': '12345'}
```

3. When the built-in function `len` is called on an object of this class, it should return the sum of two terms:
    1. `10` times the number of items in the internal `items` dictionary.
    2. The length of the file contents or recursive directory size.

    ```
    In [13]: len(d) # ONE ITEM (10) OF LENGTH 5 → 10 + 5
    Out[13]: 15
    In [14]: d.additem("docs", Directory()) # ADD BACK THE DIRECTORY
    In [14]: len(d) # TWO ITEMS (20) ONE OF LENGTH 5 AND ONE EMPTY (0)
    Out[14]: 25
    In [15]: d.items["docs"].additem("bank", Directory()) # ADDING A SUBDIRECTORY
    In [16]: len(d) # TWO ITEMS (20) ONE OF LENGTH 5 AND ONE WITH AN ITEM IN IT (10) → 35
    Out[16]: 35
    ```

4. When printed, a `Directory` object should look like:

    ```
    In [17]: d
    Out[17]: Directory with 2 items (35 bytes)
    ```

    That is, it should print the number of items in the internal dict and then call `len` on itself and print the result as its number of “bytes”.

5. Create a class called `LousyFS`, whose constructor takes a single argument `totsize`. This will be a positive integer. The constructor creates two internal attributes: a `Directory` called `root` and a `totsize`.

    ```
    In [1]: fs = LousyFS(100)

    In [2]: fs.root
    Out[2]: Directory with 0 items (0 bytes)

    In [3]: fs.totsize
    Out[3]: 100
    ```

6. Write a method called `getdir` that takes a single argument `path`, which is a list of strings. The function should start from `self.root` and navigate down the directory structure using the names in `path`, finally returning the resulting `Directory` object.

    ```
    n [4]: fs.getdir([]) is fs.root # AN EMPTY LIST JUST RETURNS root
    Out[4]: True
    In [5]: fs.root.additem("docs", Directory()) # ADD A SUB-DIR
    In [6]: fs.getdir(["docs"]) is fs.root.items["docs"]
    Out[6]: True
    In [7]: fs.root.items["docs"].additem("bank", Directory()) # ADD A SUB-SUB-DIR
    In [8]: fs.getdir(["docs", "bank"]) is fs.root.items["docs"].items["bank"]
    Out[8]: True
    ```

Tip: use an auxiliary variable initialized to `root`, and then update it with each successive sub-directory that you encounter on the list. Then return it. It's 4 lines of code.

7. Add the two methods `additem` and `delitem` to the `LousyFS` class.
   These methods will be similar to those in the `Directory` class but with an additional **first** argument `path`.

   The idea is that we are creating/deleting an object inside a particular directory, identified by the `path` list. So for example `additem(["docs"], "file.txt", "xyz")` is adding a file inside the `"docs"` subdirectory. In the method, use the path argument like this: pass it to the `getdir` method, obtaining a `Directory` object. Then call the corresponding `additem` / `delitem` method for that directory.

```
In [9]: fs.additem(["docs"], "my_password.txt", "12345") # ADD A FILE INSIDE THE docs
DIRECTORY
In [10]: fs.getdir([]) # THIS IS root
Out[10]: Directory with 1 items (35 bytes)
In [11]: fs.getdir(["docs"])
Out[11]: Directory with 2 items (25 bytes)
In [12]: fs.additem([], "pics", Directory()) # ADD A SUBDIR TO THE ROOT
In [11]: fs.additem(["pics"], "smile.jpg", ":)") # ADD A FILE INSIDE pics
In [13]: fs.getdir([]) # ROOT AGAIN
Out[13]: Directory with 2 items (57 bytes)
In [14]: fs.delitem([], "pics") # DELETE THE pics DIRECTORY
In [15]: fs.getdir([]) # BACK TO WHERE WE WERE
Out[15]: Directory with 1 items (35 bytes)
```

8. The only other thing to ensure is that there is enough space on disk when we use `additem`. Write a method called `free` that takes no arguments and returns `self.totsize - len(self.root)`. Then, in `additem`, check that the result of `free` is at least 10 plus the length of the contents. Otherwise raise an exception.

```
 In [16]: fs.free()
 Out[16]: 65

 In [17]: fs.additem([], "largefile.zip", "X" * 56) # ERROR, NOT ENOUGH SPACE ON DISK
("X"*55 SHOULD WORK)
```

9. [OPTIONAL, MOSTLY TO MAKE THE EXAMPLE BELOW NICER] When printed, a look like in this example:

```
In [18]: fs
Out[18]: LousyFS (total: 100 bytes, used: 20 bytes, free: 80 bytes)
```