Exam for the course DAT171 Object oriented programming in Python

Time: 16th August 2022 8:30-13:30

Teacher: Mikael Öhman (073-6837674)

Permitted aids: Cay Horstmann: Python for everyone. Manuals and lecture notes are available on the computers.

Teacher will visit the rooms: Around 10:00 and 12:00

Formalities: In each file you hand in, you should write your examination code as well as the computer "number".

The documentation and lectures notes are available on `C:\__EXAM__`. Verify that these files are there immediately. Handing in the code should be done under the same folder: `C:\__EXAM__\Assignments\`, when you are finished. If you do not store the files here, they are not included in the exam. Save the files for each question in the appropriate folder, e.g. `C:\__EXAM__\Assignments\Question1\`. *Make sure you only hand in one solution for each question, otherwise you will receive **zero** points.* Complex sections of your code should have a descriptive comment of what is achieved.

When you finish the exam you should log out and fill in the (empty) exam cover page like normal and hand this in at the end of the exam.

Corrections: The results will be announced through Ladok, on or before September 1st. The review will be the same day (September 1st) 12:20-13:00 at Hörsalsvägen 7b, 3:rd floor.

Grading: There is a total of 25 points which yields grades at the standard 40%, 60%, and 80% limits:

```python
def grade(points):
    if points >= 20:
        return 'Grade 5'
    elif points >= 15:
        return 'Grade 4'
    elif points >= 10:
        return 'Grade 3'
    else:
        return 'Fail'
```

Question 1: Password files (6p)

In Linux-like systems we have 2 files that contain a list of the system users and groups. These files are traditionally called `"passwd"` and `"group"` respectively.

The format of the `passwd` file (containing user information) is as

```
username:password:user_id:group_id:description:home_dir:shell
```

and the `group` file as

```
group:password:user1,user2,user3
```

Typically, passwords are not stored in plain text, and the letter `x` is used as a replacement to indicate they are stored encrypted outside the file. The description is the full name of the user.

A typical `passwd` file may look like:

```
sys:x:3:3:sys:/dev/:/usr/bin/nologin
mail:x:8:8:mail:/dev/:/usr/bin/nologin
huey:quack:500:500:Hubert Duck:/home/huey:/bin/bash
dewey:quacky:501:501:Deuteronomy Duck:/home/huey:/bin/bash
louie:quackers:502:502:Louis Duck:/home/huey:/bin/bash
gizmoduck:x:503:503:Fenton Crackshell:/home/gizmo:/bin/bash
goofy:x:504:504:George G. Geef:/home/goofy/:/bin/bash
chip:x:600:600:Chip:/home/goofy/:/bin/bash
dale:x:600:600:Dale:/home/goofy/:/bin/bash
```

and a typical `group` file as:

```
ducks:x:huey,dewey,louie,gizmoduck
super-hero:x:gizmoduck
rodents:x:chip,dale
clumsy:x:gizmoduck,goofy,dale
```

We wish to create a tool to display this data for a system administrator in a different way, so you will need to read these files and output a different format.

Part A (1p)

Write a function, `read_users` which parses the `passwd` file into a dictionary. Ignore all entries with `user_id` below 500, as these are reserved for running the system services.

We don't need all fields from either file, see below for usage:

```
>> names = read_users('passwd.txt')
>> print(names['huey'])
Hubert Duck
```

Part B (2p)

Write a function `read_groups` that reads the group file into a dictionary that maps `user -> groups`, see example usage:

```
>> groups = read_groups('groups.txt')
>> print(groups['dale'])
['rodents', 'clumsy']
```

Hint: You may find `defaultdict` *from* `collections` *helpful.*

Part C (3p)

Write a function `print_user_table(names, groups)` which prints a nicely formatted table like this:

```
User  | Name             | Groups
------+------------------+-------
chip  | Chip             | rodents
dale  | Dale             | rodents, clumsy
dewey | Deuteronomy Duck | ducks
...
```

The `User` and `Name` column should widen to automatically fit the longest user and name. It should be sorted according to the username (the first column).

Question 2: Musical analysis (6p)

You are prototyping an Python based musical tuning program for a well known Violin manufacturer. The company has sent over a test recording of a violin, see the file `violin.wav`, and your boss has given you the task to implement two parts of the project.

Note: You `must` use NumPy types to store any sequence data in this task! Also, you `must not` use any loops in this task!

Part A (2p)

Task: Implement a function `pitch` that takes a floating point value `frequency`, in the unit of Hertz (Hz), as input and returns a text string with the corresponding musical note and octave. Remember to document the function.

Example: The frequency 440 Hz corresponds to A4, were A denotes the tone and 4 denotes the octave. Here is an example usage of the function `pitch`

```python
freqency = 440. # unit Hertz
print(pitch(frequency), '=', frequency, 'Hz')
```

should give the result

```
A4 = 440.0 Hz
```

Background: Converting a frequency f in Hertz to a note n and octave o can be done this way:

- Compute $h = 12 \cdot \log_2(f/f_{C0})$
  - Here f_{C0} is the frequency for the note C0 which is given by $f_{C0} = f_{A4} \cdot 2^{-4.75}$
  - $f_{A4} = 440\,\text{Hz}$ is the base frequency
- Let $[h]$ denote the integer closest to h
- The octave o is given by the result of the integer division of $[h]$ with 12
- The index of the note n is given by the remainder of the integer division above
- To map the note index n to actual notes use the list of notes: [C, C#, D, D#, E, F, F#, G, G#, A, A#, B]

Hint: Use the standard `round` function and the `log2` function from the `math` module of the Python standard library, or the equivalent NumPy functions.

Part B (1p)

Task: Write code that reads the sound file `violin.wav` and plots the sound amplitude data as a function of time, using *seconds* as the time unit (i.e. for the x-axis). Make sure to add a title as well as x- and y-labels including information on units.

Hints:

- To read the file use a suitable method from the `scipy.io` module, see the SciPy documentation.
- The sound data has two components,
  - the amplitude $A(t)$, as a function of time t, and
  - the sample rate R, given in the unit of Hz, determining the number of amplitude data points per second.
- The time t_i of the data point with index i is given by $t_i = i/R$.
- For an example plot see Fig. 1.

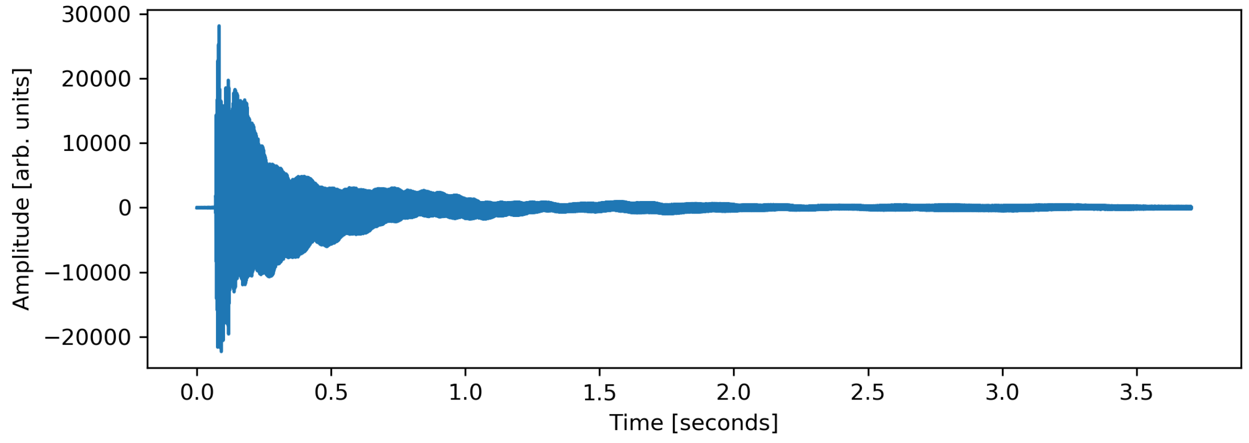

Figure 1: Example plot of the sound amplitude as a function of time (of a different sound than `violin.wav`).

Part C (3p)

Task: Determine the dominating frequency in the sound file `violin.wav` and use your `pitch` function to compute the corresponding musical note. Do this by *computing* and *plotting* the amplitude of the sound as a function of frequency (instead of time). Make sure to add a title as well as x- and y-labels including information on units. Use a suitable NumPy function to find the maximum amplitude in frequency and use this frequency as input to your `pitch` function, to compute and print the corresponding musical note.

Hints:

- The amplitude of the sound $A(v)$ as a function of frequency v can be computed by taking the absolute value of the Fourier transform $\mathcal{F}\{\cdot\}$ of the amplitude as a function of time $A(t)$, i.e. $A(v) = |\mathcal{F}\{A(t)\}|$.
- To compute $\mathcal{F}\{A(t)\}$ use the `numpy.fft.fft` function, and to compute v use the `numpy.fft.fftfreq` function, see the NumPy documentation.
- For an example plot see Fig. 2.

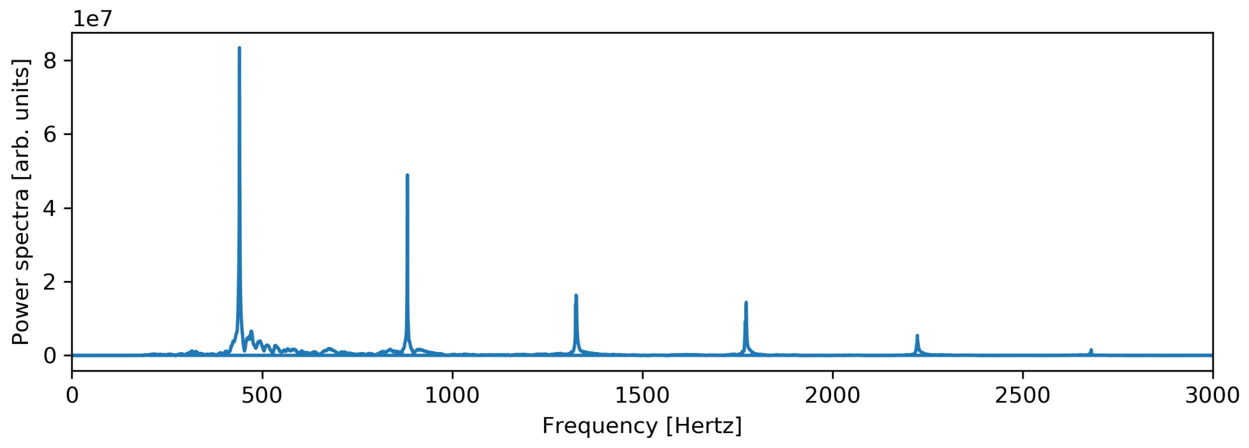

Figure 2: Example plot of the sound amplitude as a function of frequency (of a different sound than `violin.wav`). The frequency of the tone in this example is 440 Hz, which corresponds to the note A4, as can be seen in the plot from the location in frequency of the largest peak.

Question 3: Repeated list (6p)

Lists are key data-structures and there can be many useful variants for different special purposes. One such case is a list that repeats the same sequence multiple times, which we can efficiently store *without allocating additional memory* while implementing all the methods that make it look and behave just like a big list.

Part A (0.5p)

Create a new class `RepeatedList` which takes a list and the number of repetitions. This class and its methods should not consume any more memory even if the number of repetitions is large.

```
>> repeated_list = RepeatedList([1, 2, 3], 3)
```

Part B (0.5p)

Implement support for getting the length of your `RepeatedList`, e.g.

```
>> len(repeated_list)
9
```

Part C (1p)

Implement support for getting items out of the list. Raise an `IndexError` if the index is negative or out of total range.

```
>> repeated_list[2]  # third item in first repetition
3
>> repeated_list[7]  # second item in third repetition
2
>> repeated_list[9]  # Should throw an IndexError!
>> repeated_list[-1]  # Should throw an IndexError!
```

Part C (0.5p)

Add `str` support that prints output like this

```
>> str(repeated_list)
'[1, 2, 3]*3'
```

Part D (0.5p)

While debugging small problems, we find that it's useful to expand the RepeatedList into a big ordinary list. Add a method method `tolist()` that returns an expanded `list`.

```
>> repeated_list.tolist()
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Part E (3p)

Add support for iterating over your `RepeatedList`. The `iter(repeated_list)` function should return a `RepeatedListIterator` object which should keep iterating over the given list the specified number of repetitions (see loop examples).

Remember the purpose of this class: you should be able to iterate without expanding the full list.

Example:

```
print("Test [1,2,3]x3")
x = RepeatedList([1,2,3], 3)
assert len(x) == 9  # Verify true length
assert len(x.tolist()) == 9  # Verify the length of the expanded list
print(x)  # Test printing
for i, v in enumerate(x):  # Iterate over entire sequence (9 elements)
    print(f'{i}: {v}')
print(x.tolist())  # Expand the list to verify the order is as expected
assert x.tolist() == [v for v in x]  # Verify that tolist and iterating are consistent

print("\nTest [4,5,6]x0")
y = RepeatedList([4,5,6], 0)
assert len(y) == 0
print(y)  # Test printing
for v in y:
    assert False  # Should not occur, zero elements

print("\nTest []x0")
z = RepeatedList([], 3)
assert len(y) == 0
print(z)
for v in z:
    assert False  # Should not occur, zero elements

print("\nTest 3 quintillion floats!")
w = RepeatedList([3.14, 2.72, 1.41], 1000000000000000000)
assert len(w) == 3000000000000000000  # Verify the length
print(w)  # Test printing
print(w[2999999999999999999])  # Test getting items out of the list
```

Output:

```
Test [1,2,3]x3
[1, 2, 3]*3
0: 1
1: 2
2: 3
3: 1
4: 2
5: 3
6: 1
7: 2
8: 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

Test [4,5,6]x0
[4, 5, 6]*0

Test []x0
[]*3

Test 3 quintillion floats!
[3.14, 2.72, 1.41]*1000000000000000000
1.41
```

Question 4: File structure (7p total)

In this question, you will implement classes for representing a simple file structure, which would be useful when working on a library for a archive (like a zip file). For the sake of keeping the exam question reasonably short, we will not consider the actual file contents, but only the metadata (file structure, file names etc.)

You should generate some XML markup as shown in the example. (XML is simple markup language for storing and transporting data.)

Part A (2.5p)

Create a base class `Node` stores a variable `name`. This class should have:

- a method `escaped_name` that returns the name of the node but every with special characters replaced (see list below).
- an abstract method `xml_structure` that takes an optional argument `indent` with a default value of 0
- an abstract operator for `len()` that computes the number of nodes in the entire tree recursively.

Note: the name "node" is typically used in file systems for referring to an object, like a directory, file or link.

- ampersand (`&`) is escaped to `&amp;`
- double quotes (`"`) are escaped to `&quot;`
- single quotes (`'`) are escaped to `&apos;`
- less than (`<`) is escaped to `&lt;`
- greater than (`>`) is escaped to `&gt;`

Part B (2.5p)

Create a subclass `Directory` that takes a `name` and a list of `nodes` that it should contain:

- a method `xml_structure(indent)` should return a string that starts with `<directory name="escaped name here">` then lists xml structure all the contained nodes, then ends with `</directory>` The contained nodes should be indented 4 spaces more than the current indent. See the example below on how it should look.

- a method `add_node(node)` that adds a node to the directory.

- Using `len` on a `Directory` object should return the sum of the `len` of all the contained nodes + 1 (for the directory itself).

Part C (2p)

Implement a subclass `File` that takes a `name` and a bool `binary` which indicates if the file contains binary data (or just text):

- The `len` of a file is always 1.
- Implement a method `xml_structure(indent)` that should return a string that looks like:

```
<file name="escaped name here" binary="yes or no" />
```

The string should be preceded with as many spaces as indicated by `indent`. Remember a line break (see the example)!

Example

```
some_files = [File('Important data.dat', True), File('"Quotes" & jokes.txt', False)]
some_code = [File('important_python_code.py', False), File('cheat_sheet.py', False)]
code_dir = Directory('Code', some_code)
root = Directory('Directories can also have \'escaped\' characters', some_files)
root.add_node(code_dir)
print(root.xml_structure())
print("Total number of nodes in the tree is {}".format(len(root)))
```

should yield

```
<directory name="Directories can also have &apos;escaped&apos; characters">
    <file name="Important data.dat" binary="yes" />
    <file name="&quot;Quotes&quot; &amp; jokes.txt" binary="no" />
    <directory name="Code">
        <file name="important_python_code.py" binary="no" />
        <file name="cheat_sheet.py" binary="no" />
    </directory>
</directory>
```

```
Total number of nodes in the tree is 6
```