Advanced Python Programming

Assignment 5

'A Great Reporter'

In this assignment we are going to practice generating meaningful reports by using the data and accomplishments obtained from the two previous assignments ('Static analysis & Complexity' and 'Reflecting upon itself'). The outline is following: integrate the mentioned assignments into one library with some minor changes, create a simple program that use it, learn how to use `matplotlib` as a tool for generating plots and finally produce a multipage pdf file out of the data you'll get by applying own library onto your code. At the end it remains to upload two `.py` files, resulting `report.pdf` and a few supplementary files. Just before we get started let us refresh some formal arrangements.

This time you will be checked manually. No machines are gonna be involved! So take your chance to give a full freedom for your ingenuity. *Aesthetics* of your generated pdf files will be an *important* criterion for giving and taking the points! Whatever happens our settings as usual: **5 tasks** and **10 points in total**.

- Use **Python 3.8.0** (look at `pyenv` project for reliable version switching)
- Allowed libraries: `matplotlib` with `numpy` for generating PDF containing graphs, `PyFPDF` for generating PDF containing text and `PyPDF2` for concatenating them.
  (we could limit ourselves to just `matplotlib + numpy` but the former makes simple text boxes unnecessarily complex to produce)
- Upload your solution without archive. Just a bunch of files you'll be asked to create as we move along. (don't worry if you hadn't specified your name anywhere, we will identify you by getting it from moodle)
- Please, check the **Resources** first. It will help you not to get lost among the trees.
- The **deadline** is the next Monday at 3:30pm (just before the lecture)

That's that and we are ready to start!

1. Merge the codebase taken from two last assignments (probably these files would be `reflect.py` and `main.py`) and produce single-file module `Xlib.py`. Where `X` is a *meaningful* name for your brand new library. Names like `cat123.py` or `nonamelib.py` will degrade a point. We want you to demonstrate your great naming-skills! And second, do not forget that *you* are the end-user of your library.

```
$ ls assign05/
program.py
Xlib.py

$ cd assign05/
$ cat program.py
import Xlib
```

```
if __name__ == "__main__":
    # nothing useful
    # just to make sure your merge doesn't crash!
    print("hello")

$ python program.py
hello
```

2. Now, we gonna rearrange your library so that it has exactly two decorators: `@stat_object` (previously known as `@reflect`) and `@stat_complexity` (what we've got from Assignment 4). Each is supposed to produce an ordinary python dictionary that you must simply dump right into your terminal (for demonstrating purposes). Here is how:

```
$ cat program.py
from Xlib import *

@stat_object
@stat_complexity
def foo():
    print("hello")

if __name__ == "__main__":
    foo()

$ python program.py > stats.txt
$ cat stats.txt
{
    'name': foo,
    'type': <class 'function'>,
    ... (remove 'Complx' field)
    'output': "hello"
}, {
    'operators': {
        'if': 0, ..., 'N1': x
    },
    'operands': {
        'docstrings': 0, ..., 'N2': x
    }
    'program': {
        'vocabulary': x, ..., 'effort': x
    }
}
```

(Note 1: you don't need to make pretty printing, this is just *illustrative* example)
(Note 2: for the whole list of `'fields':`, please refer to previous assignments)

3. Now we're close to the culmination. Generate pdf reports! First, add two more decorators named `@report_object` and `@report_complexity` accordingly. Then apply them instead:

```
$ cat program.py
from Xlib import *

@report_object
@report_complexity
def foo():
    print("hello")

if __name__ == "__main__":
    foo()

    # Important (!)
    # There is no chance to produce any meaningful reports out of
    # this dull foo/bar functions. So be smarter and apply your
    # decorators onto themselves as we did it before:
    # report_object(report_object)(None)
    # report_complexity(report_complexity)(None)

$ python program.py
(no output)
$ ls ./
foo_object.pdf
foo_complexity.pdf
program.py
stats.txt
Xlib.py

$ pdfviewer foo_object.pdf
```

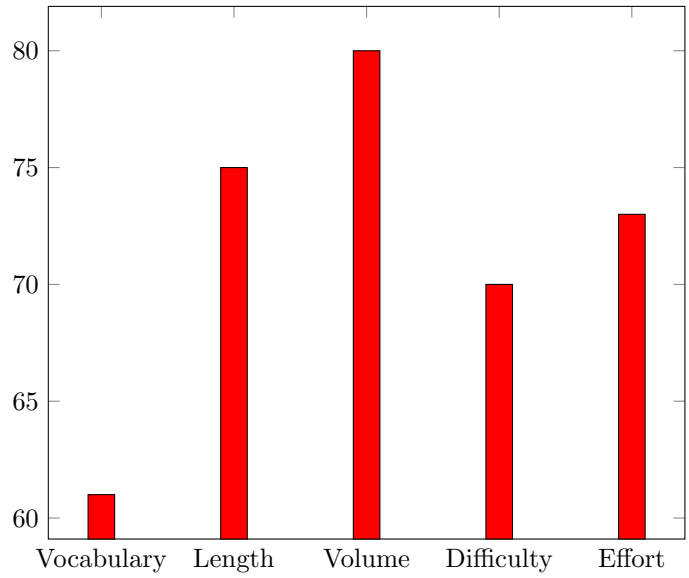
```
Use your imagination to produce a nice
and illustrative output based on your
data. It is ok here to be only textual
content without any shaped graphs since
we don't really have anything to plot out.

If possible put a page number at the end.


                                1
```

```
$ pdfviewer foo_complexity.pdf
```

```
Again use your imagination to produce a nice output.
Remember, the aesthetics will be judged.
This time the graph is a must.
```



1

4. Provide a mechanism for specifying module-wide and decorator-specific parameters. The important option will be `multipage` that tells your library to aggregate any applied report decorators into a single multipage file. Other options should affect resulting layout (format, showed sections, etc). Here is example:

```
$ cat program.py
from Xlib import *

rc(multipage=true, filename='report.pdf', papersize='a4')

@report_object(output=False)
@report_complexity(operators=False, operands=False)
def foo():
    print("hello")

if __name__ == "__main__":
    foo()
    # do not forget what has been said previously on
    # alternative approach of invoking decorators

$ python program.py
(no output)
```

```
$ ls ./
foo_complexity.pdf
foo_object.pdf
report.pdf
program.py
stats.txt
Xlib.py
```

(Note 1: `rc` is an old Unix convention for naming startup scripts and configs)
(Note 2: implement *at least* the parameters listed above: `output=False` means do not put program's stdout into report, `operators=False` do not put operators' statistics into report, and so on)
(Note 3: consider the listing above as files required for upload. But it might be not *exactly* the same. For example, instead of `foo_complexity.pdf` where `foo` is an object name and `complexity` is decorator's basename, you may get `report_complexity_complexity.pdf` if you apply decorator onto itself)

5. Let your decorators to work with classes! What the fields to plot out in addition to or instead of existing ones is up to your ingenuity! No examples will be given. Keep your imagination open and adapt what feels appropriate to you.

Resources

1. Gentle introduction to `matplotlib` 1, 2
2. How to create bar charts 1
3. How to produce text-only boxes instead of graph 1
   (please do use the monospace font when printing the stdout)
4. How to use multiple plots 1
   (good to know before you start producing multipage files)
5. How to produce multipage PDF file 1