The exam

This document describes the idea behind the test for the module 1 of the Software and Computing course, and the module 2 for the Applied Physics branch of the same course.

general guidelines

- You will be evaluated on a programming project;
- this project needs to be hosted on a public repository;
- you are free to choose the programming language of the project;
- accepted control version systems are git and fossil;

topics of evaluation

The project must be executable on a different machine from the one you have developed it into. Limitations can be put on the operative system and platform version required, but if so they must be explicitely specified.

- clarity of the repository commit history
- organization and clarity of the source code
- organization and clarity of the documentation
- presence and executability of test routines

optional bonuses (don't rely on them)

- Usage of innovative technologies and libraries
- Contribution to open source projects

exam progression

Please start discussing your exam project **before** starting to work on it. Discussing it early will avoid wasting effort and time for something that is not appropriate for the exam!

possible projects:

- an idea that you find interesting (please discuss it first)
- deciding a project together
- Sharing the project with other exams or thesis

The spirit of the exam

The goal of the exam IS NOT to assess if you can write a very complicated simulation.

The goal is to show that you are able to write a software that can be used by other researcher and that it can pass the "bus test": if tomorrow you are unable to work on this project, would other people be able to keep it alive?

To fullfill this goal, it is not useful to perform virtuosisms of computational skills. The project does not have to be complicated. It has to be reliable and understandable by others.

This other person should not be burdained of effort to understand your code.

"Go and read the source code" it's not a reasonable answer.

"Go and edit the source code" to use it is even less so.

They should be able to install your code, run it and replicate your results.

consider giving the software a command line interface that can be run with some parameters, rather than having to explicitely run a bunch of scripts in a non explicit sequence.

General guidelines on the various topics of evaluation

Documentation

Documentation should be written in English, with proper sentences and coherent structure. The whole documentation should be accessible from the landing page of the repository, either in the readme or using hypertextual links: the user should not be left guessing on where to find it.

The documentation should explain:

- what problem is the function or program trying to solve
- what are the parameters that are exposed to the user and, if there are defaults parameters, what they are
- how it is implemented (does it use specific algorithms? do have any peculiar assumption about the input data?)

If possible the documentation should also include examples of execution, configurations and results, so that the user can understand the basic uses of the program/library, and could test on their own.

a good reference for writing documentation is
https://www.divio.com/blog/documentation/

Testing

Testing should involve all the functions exposed to the user, and should ideally cover any combination of parameters. This means that in general is a good idea to keep the functions simple and composable, to avoid overly brittle and complicated test suites.

The test name should be indicative of what the test is trying to achieve. `test_1` is not a clear name `test_function_is_even` is more clear.

Is possible, write documentations (docstrings) for your tests as well, following the principle of explaining WHY the test is being performed, alongside WHAT is testing.

REMEMBER THAT FLOATING POINT NUMBERS HAVE ISSUES WITH EQUALITY TESTING!!!

Make sure that any test could only fail for a single reason: if a test have several failing conditions, it will be difficult to pinpoint the exact cause.

Every time you find a bug, write an associated test **before** solving the bug, make sure that it fails and then solve it (making it sure that now it passes). This will prevent the bug from showing again in the future.

Tests are a way to also establish the interface of your program. They are a promise to your users: if something is tested it should work. This means that every commit should be tested and only submitted if **all** the tests passes.

Even if not compulsory, I strongly suggest to use a coverage test system (with libraries such as *coverage.py*) and try to achieve 100% coverage.

test code should be well separated from the library (or simulation) code, and definitely not mixed in the same file.

testing does not need to verify that the code can be run (this should be taken for granted), but that upon executino the results are the one expected. This is the reason to use asserts.

visualization functions - no testing needed

Your code might contains faunctions that, given some data in input, will generate some visualization of those data.

In general, it's accepted that this kind of functions don't require testing, as the effort necessary to have a robust test will overwhelm any benefit.

Of course, a function of this kind should not contain ANY PREPROCESSING of the data, but just the visualization itself.

Testing functions and printing

Testing function should not print or return anything, as this would only confuse the result of the testing suite.

Use of Hypothesis

The use of the hypothesis library is welcome, but only once there is a proper test suite in place with more simple tools such as unittest or pytest

Code organization and clarity

Follow a consistent programming style. A good starting point is the PEP8. There are many guidelines out there, pick one and follow it.

Use proper naming, both explanatory and consistent. This helps the user using your library. If in one part of the library you have functions such as `has_property`, don't change the style in another part to something like `is_something`

Avoid dependency from global states as much as possible. The only tolerated exception is for configuration values that are never modified. Do not write on the global state. No, yours is not a special enough case.

leverage the power of functions to avoid useless code repetition and make your code easier to read for someone new to it. Functions should ideally do a single thing: huge number of parameters in the call is usually indicative that the function is trying to do to much.

functions should be decorrelated, meaning that there should not be a fixed order to call them otherwise weird error arises. If a group of functions needs to be called in a fixed order, you are probably trying to write an object without realizing it. In some cases this can be simplified using *partial* functions and *contextmanager* decorators.

If your code dependes on external libraries, remember to point out which ones, and which is the minimum (and possibly maximum) version that are necessary. In general try to balance out how many libraries your code dependes on. Some libraries are commonly used by most users in the same community, and those are not a problem (such as numpy, scipy, pandas and scikit.learn for the scientific python community), but try to limit dependencies from *unorthodox* libraries

if you need to connect to online services, consider *caching* or, at least, minimize the number of connections used to avoid bogging them down.

Make sure that there are no reference to specifi paths of your computer in the code! Asking the library to load a file in the directory `\home\myname\myprojects\SandCexam` is unacceptable. It is suggested to use a third party continuous integration service such as TravisCI, as it will show most of these brittle errors. I need to be able to run the software on my computer, this is a prerequisite, and as such this kind of mistake precludes the project to receive and evaluation!

comments

Remember to comment the code: comments should indicate WHY the code is doing something, not describe WHAT is doing. Proper function and variable naming can help reduce the needs for comments.

bad:

```
lenght = min(lenght, 80)
```

better:

```
# limit the text lenght to avoid overflow issues
lenght = min(lenght, 80)
```

best:

```
# limit the text lenght to avoid overflow issues
terminal_character_limit = 80
text_line_lenght = min(text_line_lenght,
                       terminal_character_limit)
```

assert and exceptions

In python don't be overly defensive with testing the input for a function. If your function need to insert some values in a dict, you don't need to test that the input is a dict, but only that is a mappable (does have a `set_item` function).

Even better, don't try to verify beforehand but rather use a try except clause in the code and explain the problem to the user.

Don't use `assert` to check for important conditions, but use if-else clause and raise an approriate exception (subclassing one of the existing one, if needed) if the condition is not valid.

Asserts are supposed to be used at the end of the code to make the code invariant explicit (for example that the two returned arrays are of the same lenght). In a reasonable condition one should never expect to see one of these `assert` fail: they are used to communicate infomations to the reader.

bad:

```
def add_key(mappable, key, value):
    assert isinstance(mappable, dict)
    mappable[key] = value
```

better:

```
def add_key(mappable, key, value):
    if not isinstance(mappable, dict):
        raise TypeError("the mappable object should be a dict")
    mappable[key] = value
```

even better:

```python
def add_key(mappable, key, value):
    try:
        mappable[key] = value
    except TypeError as e:
        err = "object does not support item assignment"
        if err in e.args[0]:
            msg = "{} doesn't support item assignment"
            msg = msg.format(type(mappable))
            raise TypeError(msg) from None
```

best:

```python
def add_key(mappable, key, value):
    mappable[key] = value
```

Version control commits

DO NOT UPLOAD PASSOWORDS OR ANY OTHER SECURITY INFORMATION!!! This includes also the ip address of servers, usernames, personal informations or anything of the sort. Include them in a separate text file if you really need them and only include in the commit a dummy file with fake informations.

All commits should have a brief and descriptive title. "some edits" is not a good description.

Each commit should be a single kind of edit. Editing a function might need to update also the documentation and testing, and possibly other code that relies on that function. But don't mix random extensions of the documentation, code editing, and so on.

try to avoid including temporary files (such as the `__pycache__` folder) in the commits, as they make the repository bigger and harder to understand.

avoid including very big files in the repository, unless necessary. Consider the possibility of dynamically generate them during the installation or the first run.

Having multiple smaller commits is absolutely ok. Projects with hundreds of commits are absolutely normal, don't be afraid to commit multiple time a day.

Do not upload datasets, especially if they are from an experiment that you're collaborating with, as you might have copyright and intellectual property issues.

Kind of possible projects and gotchas of each one

- libraries
- simulations

these are not the only possible ones (frameworks, web services, etc... would be accepted upon discussion), but they are the vast majority of the projects.

management of simulations

If the project is a simulation, the main goal is to be able to replicate the results, and to batch execute them.

- there should be no hard dependency on pathways or files from your computer
- random elements should be managed explicitely using a random seed

- the simulation, analysis and plotting steps should be separated, so that the analysis can be repeated without performing the simulation again. Ideally should be possible performing batch analysis of the results of several simulations.
- different configurations for the simulation should be included in a separate configuration file, that could be provided by the user

management of libraries

if the project is a library, the main goal is for a user to be able to use it in their own libraries and simulations.

- the API of the library (i.e. the functions and how they are related) should be properly designed and explained
- include examples and tutorial in the documentation

- configuration of the library should be transparent to the user, and should not rely on weird system states
- be clear on the algorithms used and possible limitations of the method (memory, disk space, parallelism, etc...)

Jupyter notebooks

Do not use Jupyter notebooks to store functions, classes and testing.

Palce them in separate files and import them in the notebook.

Jupyter notebooks are not designed for lont term, clean code storage, but rather to keep trace of the execution of said code. They are ideal for tutorial, how-to and similar, and their use is welcome to show an actual execution of your code.

examples of repositories from other students

these are some repositories that obtained the full mark in the exam (for the applied physics course), and could be used as a reference.

- https://github.com/JonathanFrassineti/Software-Project
- https://github.com/filippocastelli/KEGGutils

- https://github.com/riccardoscheda/AnomalousDiffusion