Object Oriented Programming

Practical Assignment 1

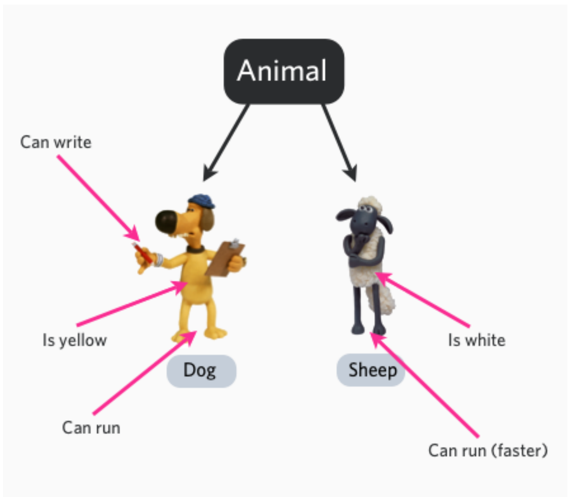
Informatics 2 for Biomedical Engineers

Graz University of Technology

Abstract

Programming Languages provide a lot of freedom to developers who implement a piece of software. However, this freedom can result and code that is not reusable or requires a great deal of maintenance. Object Oriented Programming (OOP) is the widest spread and most common approach for software-engineering tasks. The fundamental concepts of this programming method are classes and objects. These classes can contain data and functions and enable the developer to build modular reusable software. Python heavily supports this approach and even supports ideas impossible to many other languages.

To become familiar with the concept the first practical assignment is to implemented the parts of the programs from Informatics 1 in an object oriented fashion.



Object Oriented Programming. A simplification
http://www.codercaste.com/2011/01/12/what-is-object-oriented-programming-and-why-you-need-to-use-it/

1. Tasks

This assignment iterates on the assignments from informatics 1. However, now the task is to build a package using object oriented code. For further information and reference solutions please refer to the Wiki[1]. **IMPORTANT:** If you utilise the reference solutions or any other code taken from elsewhere please follow the citation rules (Section 7)!

For students who did not participate in informatics 1:
For instruction on how to set up python on your system please refer to Appendix A. How to obtain the data files can be found in Appendix B

1.1. Class Implementation (3pt)

Create the files and classes according to the specification. Make sure to use pythonic getter and setter functions. Additionally think if every member variable needs a getter or setter. You should create the functions for every single member variable you have, but when you do not allow it to be set from outside your setter should block this action. Follow the naming convention for all variables and functions! This classes

1.2. Abstraction (2pt)

The InputParser Class is abstract and only provides some basic structure. How the input file is read is implemented in the PtbdbParser class. Use proper abstraction when implementing this.

1.3. Pythonic Getters and Setters (1pt)

Getters and setters in python can be done differently to other programming languages. Use @-decoraters to implement them. See the documentation for more information on this. [2]

1.4. Documentation (2pt)

One key aspect of object oriented development is to account for code-reuse. Be it by yourself or somebody else. Understanding undocumented code can be very difficult. Write docstrings for each class and method. Use triple-quotes `"""` when writing them. Docstrings are technically comments so the interpreter will completely ignore them (unless you specifically ask it

[1] `https://palme.iicm.tugraz.at/wiki/Info1BM`
[2] `https://docs.python.org/3.5/library/functions.html#property`

to). But development software often attempts to read and understand them. Whilst we do not strictly enforce the standard (PEP-257) your doc strings should explain what your class or function does and what it returns. For further information, read the PEP-257 specification. [3]

Example for a docstring:

```python
def multiply(x, y):
    """
    Multiplies the value of x by the value of y and returns the result
    """
```

1.5. Proper Typechecks and Error Handling (1pt)

Any time you handle parameters that come from outside of your package you should check them if they are the correct type the typical functions python provides are "isinstance()" and "type()". Note, your should also always perform checks when you are handling files.

1.6. Creating a python package (1pt)

Python packages come with an initialisation file called __init__.py. Here you can define which classes should be exposed when someone imports your software. Refer to the package structure below to see which components of your package are supposed to be public.

2. Package Structure

The program you develop consists of exposed (usable by others) and hidden components. See Figure 1 for a visualisation. Note, python does not necessarily stop programmers from using the classes you did not intent for public use. But you can somewhat control the default behaviour using the __init__.py in your package.
Your package should (in addition to the initialisation file) consist of these 5 files/classes:

1. InfoBM
2. InputParser
3. PtbdbParser

[3] `https://www.python.org/dev/peps/pep-0257/`

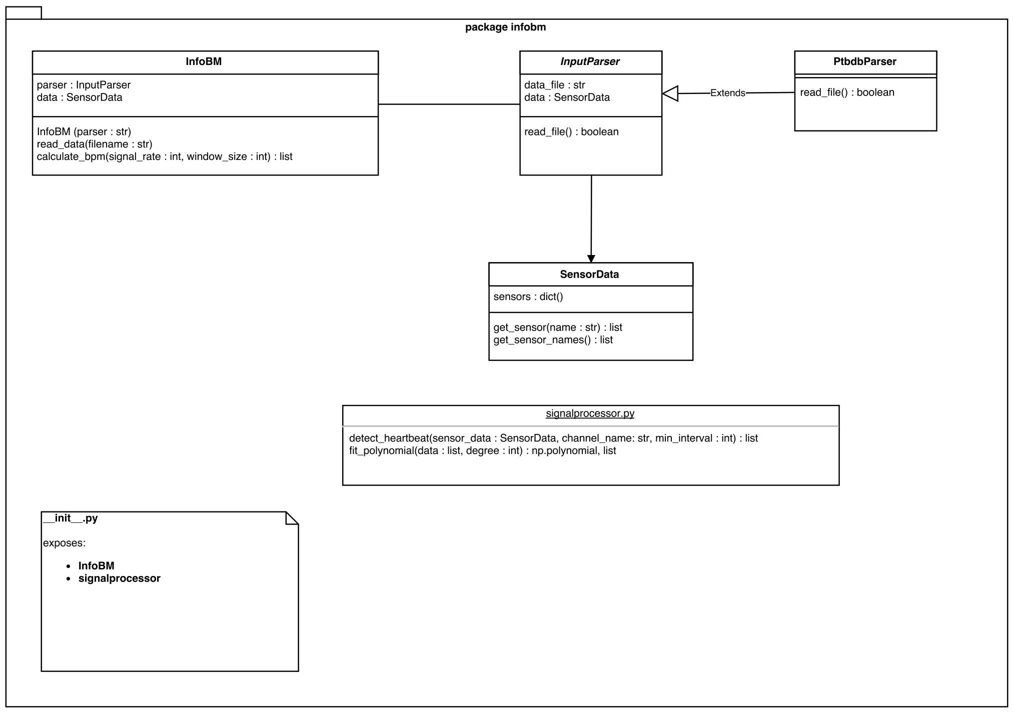

package infobm
InfoBM
parser : InputParser
data : SensorData
InfoBM (parser : str)
read_data(filename : str)
calculate_bpm(signal_rate : int, window_size : int) : list
InputParser
data_file : str
data : SensorData
read_file() : boolean
Extends
PtbdbParser
read_file() : boolean
SensorData
sensors : dict()
get_sensor(name : str) : list
get_sensor_names() : list
signalprocessor.py
detect_heartbeat(sensor_data : SensorData, channel_name: str, min_interval : int) : list
fit_polynomial(data : list, degree : int) : np.polynomial, list
__init__.py
exposes:
InfoBM
signalprocessor

Figure 1: **Class Diagram.** This figure shows the class structure of the program.

4. SensorData
5. SignalProcessor

3. Classes and Purpose

3.1. InfoBM

This is the core class of your package. When a developer imports your package they should normally create an object of this class. The constructor of this class has 1 parameter. This string should decide which parser is used (implement this as if/else). Since we only have the parser for ptbdb data the only valid value for this at this point would be *"ptbdb"*. You should note this in your documentation! In addition to the parser this class also holds a SensorData object (see below). The class provides 2 other functions. First, *read_ data()* which calls the similarly named functions of the selected parser and if everything went well (parser returned a *True*) it fetches the SensorData object from the parser and stores it in the member variable. Second, *calculate_ bpm()* uses the BPM calculation method from last semester [4]. The parameter signal rate should define how many signals per second should be assumed for the calculation. The second parameter defines the window size in *Number of Signals*. Note, when you use the reference implementation you need to change the calculation!

3.2. InputParser

This class is the abstract reader class, implying nobody except its children should actually create and instance of it. However, we still define some functions and variables there so subclass can use them without specifically defining them. This class should have 2 member variables. First, the name of the input file and second, a SensorData object (or "None" before you actually have read any data). You can use the abstract read_file() function to check if the requested file actually exists and if the file name was not set. This function should return a boolean indicating whether or not everything went well.

3.3. PtbdbParser

This class is an implementation of the InputParser Class. Here we actually read the binary input data with the read_file() function. You can use the reference implementation from last semester[5]. When the file was read successfully create a SensorData instance with the data and return *True*. In case anything went wrong output an error message explaining what went wrong and return *False*. The names for the individual sensors ("i", "ii",...) can be written directly into the code!

3.4. SensorData

The SensorData class contains the data read by the parsers. Its member variable data contains the read data from the binary file. The only functions this class is a function that returns the sensor values from one specific lead, and a function returning the names of all sensors available.

[4] `https://palme.iicm.tugraz.at/wiki/Info1BM/Assignment_3`
[5] `https://palme.iicm.tugraz.at/wiki/Info1BM/Assignment_1`

3.5. signalprocessor.py

This is not a class but rather a collection of functions. Note, users are supposed to use these functions so you need to check the input if its the correct type. The file should contain 2 functions. You do not need to create a class in this file. Simply define your functions!
detect_heartbeats uses the heartbeat detection from last semesters assignment 2[6] and returns the indices of every detected heartbeat. Note, the third parameter (*min_interval*) defines the minimum distance between two heartbeats (this extension was added in assignment 3 of informatics 1). Again, feel free to use the reference solutions[7][8]
fit_polynomial() calculates a polynomial of n-th degree (n is defined as parameter) over a list of sensor values. This function returns 2 values as a tuple. First, the polynomial itself (use the array numpy returns you on polyfit()) and second, a list of values where you evaluate your polynomial from 0 to the length of the input list.

4. Exposed Functions & Freedom of Implementation

The described structure of the package is a suggestion. If you wish, you can implement the functionality in a different way. The user (who eventually uses your package) will only ever work with some of your functions. In these functions you will have to adhere to the exact definitions for names, parameters and return values. These are the elements the users will work with:

[6] `https://github.com/pkasper/info1-bm/raw/master/assignments/assignment_2.pdf`

[7] `https://palme.iicm.tugraz.at/wiki/Info1BM/Assignment_2`

[8] `https://palme.iicm.tugraz.at/wiki/Info1BM/Assignment_3`

InfoBM()

Parameters: (string) parser
Returns: This is the __init__() function of your class. The parameter is the name of the parser to use. Since we only implement a single specific parser, the only acceptable value for this parameter is "ptbdb"

InfoBM.data

Parameters:
Returns: (Sensordata) data

The user will retrieve the data object to use with your signalprocessor functions. Takes no parameters and returns the SensorData object. Implement this as a getter and throw an exception when the data is not ready yet (reading not done or never instructed to read a file).

InfoBM.read_data()

Parameters: (string) filename
Returns:

Takes the file-path as parameter. No specific return value is defined. You can return a value to confirm success or whatever you like. You can also hard-code the sensor names when you read the binary file (put them directly into your code)

InfoBM.calculate_bpm()

Parameters: (int) signal_rate, (int) window_size
Returns: (list) average bpm per window

Calculates the BPM on a list of heartbeats. The first parameter is the signal rate (ptbdb uses 1000 signals per second). The second data is the amount of signals to consider when calculating the average. Last semester this window was 10 heartbeats. You will need to adapt your calculation! The return value of this function is a list of the heartbeat for every window. This means for the first x (given via the parameter) entries you cannot calculate the average bpm. You can either put a 0 for those windows or simply return a list that misses entries for those indices. For example: Suppose you have 40000 entries in your sensordata object and you want a window size of 10000. You either return a list where the first 9999 values are 0, or you return a list with 30001 entries.

SensorData.get_sensor()

Parameters: (string) name
Returns: (list) sensor values for the requested channel

Returns the list of values for the requested lead/channel. The parameter is the name of the channel (for example "i").

SensorData.get_sensor_names()

Parameters:
Returns: (list) list of sensor names

Returns all available sensor names as a list. If you store the sensors as dictionary you can use the dict.keys() functionality to retrieve the names. Note, .keys() returns a dictionary-specific object. To retrieve a normal list wrap it with a list().

signalprocessor.detect_heartbeat()

Parameters: (SensorData) sensor_data, (string) channel_name, (int) min_interval
Returns: (list) the indices where a heartbeat was detected

Uses the threshold based heartbeat detection ($periodaverage + 2 * std$) The parameters are a SensorData object (the user will have to retrieve that from its InfoBM object), the name of the wanted signal and the minimum heartbeat distance in signals. You return a list of the indices of the detected heartbeats.

signalprocessor.fit_polynomial()

Parameters: (list) data, (int) degree
Returns: This function combines two numpy functions. The parameters are a list of values (for example: the signal data of sensor lead 'ii'). The second parameter is the degree of the fitted polynomial. Feel free to call the numpy functions in your code! This function returns two values (watch for the correct order!). First, the polynomial and second the evaluation the polynomial at each index in your list.

5. Additional Files

You can supply one additional file called *assignment_1.py* in your submission where you demonstrate the usage of your your package. We will not grade this file but they it can be helpful if we are unable to understand your code.

6. File Headers

All python source files (apart from the __init__.py) have to contain a comment header with the following information:

- Author: – Your name
- MatNr: – Your matriculate number
- Description: – Purpose of the file
- Comments: – Any comments as to why if you deviate from the task or restrictions

Please feel free to copy and paste the example and change its contents to your data. (Depending on your PDF-viewer you may have to fix the whitespaces)!
Example:

```
######################################################################
# Author:      Patrick Kasper
# MatNr:       0730294
# Description: The main file. Here we import the package and do stuff
# Comments:    This is the example comment. I just made it a bit
#              longer so it spans across multiple lines.
######################################################################
```

7. External Code

If you use **any** code from any online source you must always make a note about this in the code. A short comment explaining where you took the code from and what it does is enough. Please note, this includes and code you take from the reference solutions of informatics 1. Else these lines of code will be considered plagiarism! Remember however, this is still a 1-student assignment so do not copy the code of your colleagues!

8. Restrictions

- Do not add any bloat to your submission
- Use built-in functions where possible
- Try to avoid *.__MACOSX* or *.DS_STORE* folders in your submission archives.

8.1. Allowed packages

Any that came bundled with anaconda! See this list: `https://docs.continuum.io/anaconda/pkg-docs#python-3-6`. Any packages marked as *In Installer* can be used. Note, this link leads to the 3.6 specifications. This is intended as 3.5 is technically outdated. However, it is safe to assume everything that is listed in the 3.6 package is also available in 3.5.

9. Coding Standard

For this lecture and the practicals we use PEP 8 [9] as a coding standard guideline. Please note that whilst we do not strictly enforce all these rules we will not tolerate messy or unreadable code. Please focus especially on the following three aspects.

Spaces, not tabs. Indent your files with 4 spaces instead of tabs. PEP 8 forces this in python 3 (whilst allowing more freedom in python 2). Most editors have an option to indent with 4 spaces when you press the tab button!

Descriptive names. Use descriptive names for variables where possible! Whilst a simple *i* can be enough for a simple single loop code can become very messy really fast. If a variable has no purpose at all use a single underscore (_) for its name.

Variable Names in classes. Stick to the naming convention to define the purpose of variables in classes.

- public: no pre- or suffix (you should not have any)

[9] `https://www.python.org/dev/peps/pep-0008/`

- private: leading underscore (_) after the "self"
  - Example: self._my_variable

Max 72 characters. Do not have lines longer than 72 characters. Whilst PEP 8 allows 79 in some cases we ask you to use the lower cap of 72 characters per line. Please note that this includes indentation.

10. Automated Tests

We will test your submissions automatically. The tests will cover all functionality of the exposed classes and files (InfoBM, SensorData and signalprocessor.py).
We will monitor the standard output. For this task please make sure you do not have any other output and you use proper capitalisation (as defined by the task).
Please make sure your submission follows all the restrictions defined in this document. Your program will have a limited runtime of 2 minutes. If your submission exceeds this threshold then it will be considered non-executable. Please note that this is a very generous amount of time for any of the tasks in this course.
If your submission fails any of the automated tests we will look into your code to ensure the reason was not on our end and to evaluate which parts of your code are correct and awards points accordingly.

11. Submission

11.1. Deadline

2nd of April 2017 at 23:59:59.

Any submission handed in too late will be ignored with the obvious exception of emergencies. In case the submission system is globally unreachable at the deadline it will automatically be pushed back for 24 hours.
If for whatever reason you are unable to submit your submission on time please contact your tutor as soon as possible.

11.2. Uploading your submission

Assignment submissions in this course will always be archives. You are allowed to use .zip or .tar.gz formats.
In addition to your python source files please also pack a *readme.txt*. The file is mandatory but its contents are optional. In this file please write down how much time you spent on the assignment and where you ran into issues. This is important to us as immediate feedback so we can adjust the tutor sessions. Please note that all submissions will be anonymised prior to being read. Hence please feel free to be honest (but do stay polite).
Upload your work to the Palme website. Before you do please double-check the following aspects:

- File names and folder structure (see below)
- Comment headers in every file
- Coding standard upheld

11.3. Your submission file

```
assignment_1.zip (or assignment_0.tar.gz)
    assignment_1.py (optional / will not be graded)
    readme.txt
    infobm
        __init__.py
        infobm.py
        input_parser.py
        ptbdb_parser.py
        sensor_data.py
        signal_processor.py
```

As mentioned above this assignment provides some freedom of implementation (cf. Section 4. Thus your folder stucture might be different. Please note that with the exception of the readme.txt and your own testing and demonstration file (assignment_1.py) everything has to be inside the package folder called *infobm*.

Appendix A. The Anaconda framework

One problem with interpreter based programming languages is that you have to trust the interpreter on the executing machine to do the right things. Whilst this is manageable for the interpreter alone it becomes an order of magnitude more difficult when you use a plethora of 3rd party packages. (Such as numpy or mathplotlib) The Anaconda platform attempts to tackle this very problem.
Built specifically for data science the anaconda platform is a single installer available for all common operating systems that installs python environments with all common packages. You can download the full package here: `https://www.continuum.io/downloads`
Additionally it installs a few other very useful tools. Both *pip* and *easy install* allow the installation of various packages. Additionally a public repository for pre-compiled binaries is available. If you want to have multiple separate python installations Anaconda can set up multiple environments with different interpreter/package configurations for each. Using these features maintaining a clean python setup is a manageable with limited effort.
Please note that this is just a short excerpt of the Anaconda features for the scope of this course and the platform can do a lot more.

Appendix B. ptbdb Data

Taken from Assignment 1 of informatics 1
The data provided for this course are from the The PTB Diagnostic ECG Database[10].
Download URL:
`https://www.physionet.org/physiobank/database/ptbdb/`

Since we use real world data you can download the direct source. Feel free to implement your program with any patient data set. We will be testing your program with *patient001*, specifically, the *s0010_re* dataset. We have uploaded this set to our github. You can also pull it from there![11]

[10] `https://www.physionet.org/physiobank/database/ptbdb/`
[11] `https://github.com/pkasper/info1-bm/tree/master/assignments/data`

Whilst this may not be interesting right now playing with various datasets may yield interesting results in future tasks!
In your package you will find 3 files.

< dataset_index > .dat contains the binary sensor information. This is the file you will be working with. 12 common sensor leads are encoded frame by frame. Meaning you will get the first 2 bytes for each sensor (frame 0) before the second frame starts once again with lead 0 (*i*).

< dataset_index > .hea is the header file describing how to read the data.

< dataset_index > .xyz . These are sensor data for the 3 Frank lead ECGs. We will not be using them this course! The file is added purely for the sake of completeness!

In order to be able to read the input data you will have to look at your header file (.hea). For the task at hand only the first line is important.

Appendix B.1. Header file example

```
s0010_re 15 1000 38400
s0010_re.dat 16 2000 16 0 -489 -8337 0 i
s0010_re.dat 16 2000 16 0 -458 -16369 0 ii
s0010_re.dat 16 2000 16 0 31 6829 0 iii
s0010_re.dat 16 2000 16 0 474 4582 0 avr
s0010_re.dat 16 2000 16 0 -260 11687 0 avl
s0010_re.dat 16 2000 16 0 -214 -16657 0 avf
s0010_re.dat 16 2000 16 0 -88 -12469 0 v1
s0010_re.dat 16 2000 16 0 -241 5636 0 v2
s0010_re.dat 16 2000 16 0 -112 -14299 0 v3
s0010_re.dat 16 2000 16 0 212 -17916 0 v4
s0010_re.dat 16 2000 16 0 393 -6668 0 v5
s0010_re.dat 16 2000 16 0 390 -17545 0 v6
s0010_re.xyz 16 2000 16 0 -3 -13009 0 vx
s0010_re.xyz 16 2000 16 0 120 7109 0 vy
s0010_re.xyz 16 2000 16 0 -18 -1992 0 vz
```

The first line states $s0010\_re$ 15 1000 38400. The first number is your patient number used to identify the files. The second number is the amount of leads present in the data. 1000 frames are recorded per second and 38400 frames were recorded in total ($38.4 seconds$). The last value may be different in your data. Since we use real word samples not all ECGs are of the same length.

The next lines show the individual leads and the order how they are stored in the file. The pre-defined data-structure is ordered in the same way so you do not need to worry about that. The first value marks the file the data is in, the second shows the size of each recorded value. 16 means you are dealing with 16-bit (2-byte) entries. Thus, you want to read 2 bytes per value. (2-byte numbers are often referred to as *short*) Everything else should be ignored for this course! Please note that we will only be working with the 12 conventional leads which are all stored in the .dat file and note the 3 Frank leads (stored in the .xyz). Thus, you only have 12 values in your data structure!

Appendix B.2. Reading the Binary Data

The individual elements of the data files should are 16 bit. At the end the length of each of your *data* elements in the sensor_data should should be equal to the total amount of frames you read (38400 in our example case). Whilst there are other ways to read binary data we suggest using the *struct*-package (import struct). Please refer to the documentation [12] how to use the unpack function. (Hint: Check the format characters section in the documentation and you will want to read signed shorts! Also this function always returns a tuple and you will only need the first element.)
The data in these files are stored in a parallel format. This means you get the sensor values for the first frame from all 12 sensors followed by the second frame for, again, all 12 sensors. This is important for your implementation!

[12]`https://docs.python.org/2/library/struct.html#struct.unpack`