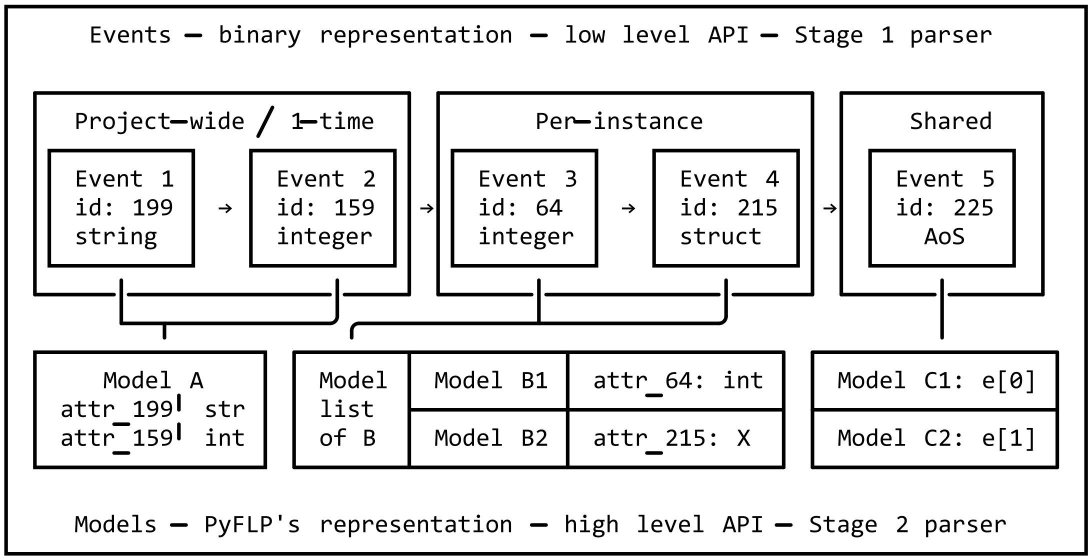
Part II: How PyFLP works

> 💡 You should read Part I before this.

PyFLP's entry-point `pyflp.parse()` verifies the headers and parses all the events. These events are collected into an `pyflp._events.EventTree`.

Schematic diagram



PyFLP provides a high-level and a low-level API. Normally the high-level API should get your work done. However, it might be possible that due to a bug or super old versions of FLPs the high level API fails to parse. In that case, one can use the low-level API. Using it requires a deeper understanding of the FLP format and how the GUI hierarchies relate to their underlying events.

What it does?

In a nutshell, PyFLP parses the events and creates a better semantic structure from it (as shown in the above diagram; stage 2 parser). I call this a "model".

Model

A model acts like a "view" or alternate representation of the event data. It has no state of its own and its composed of descriptors which get and set values from the events directly. A model is essentially stateless.

This has some advantages as compared to stateful models:

1. The underlying event data and the values returned from the model descriptors *i.e. its attributes or properties* always remain in sync with each other.
2. Since modifying the event data at a binary level means conforming to the various size and range limits imposed by C's data types, it can act as basic validation for no extra cost or implementation.
3. Avoid the use of private members in the models itself. Private members maybe a good idea in languages which have better implementation of such concepts, but in Python its quite as good as shooting yourself in the foot. Due to Python's do-whatever-you-want nature, it can lead to some very bad coding practices. This is one of the big reasons why PyFLP underwent a rewrite.
4. Nothing is done in class constructor, so if a particular set of events are out of order or follow a sequence not yet understood by PyFLP, they will fail only for the attributes which use them. Hence, what is *parseable* can still be parsed. This lazy evaluation can be good and bad both, but with adequate unit tests its more good than it is bad.

Creating a model involves a good amount of reverse engineering and insight. The models PyFLP has are based as close to the GUI objects inside FL Studio. For e.g. a pattern is represented by `pyflp.pattern.Pattern`.

A model is constructed with events it requires and additional information (like PPQ) its descriptors might need.