Class Programming Guidelines

Object-orientation in Python is more akin to that of C++ than that of Java. For example, what is an interface in Java is an abstract class with no properties in C++ or Python.

The “interfaces” are not limited to methods- public properties are also part of the interface.

Mutability

OpenLane 2 classes are based on the principle that objects that are passed between contexts are either **immutable** on termination of construction or **replicably modifiable**.

Immutable objects are just as described: they cannot be modified in-place. They may be updated only by creating a copy. This is to prevent surprises resulting from referential passing endemic to imperative and object-oriented programming.

They may, however, offer any number of read-only functions for convenience.

On the other hand, the replicably modifiable objects are those that handle non-trivial computation, e.g. Steps or Flows. Classes in this hierarchy may have a maximum of **one** public modifier adhering to these properties:

- This modifier may alter one or more properties of a class
  - The modifier shall not depend on the initial values of any altered property and may overwrite them if called again
  - If called again, it is expected to return the same result (within reason- not every aspect can be controlled, for example external filesystem modification and/or timestamps.)
- This modifier may depend on any number of private or internal modifiers
- This modifier's implementation shall be split into two:
  - A **public** part that is marked `@final`, i.e., it is not overridable. This will handle validation of inputs and outputs and thus must not be left to the whims of subclassers. It is responsible for calling the the internal part.
  - An **internal** part that is freely subclassable, however, it cannot be called from outside the public part.

For `openlane.flows.Flow`, for example, the public and internal parts are `openlane.flows.Flow.start()` and `openlane.flows.Flow.run()` respectively.

Access Control

Python has no access control, subscribing to the notion of "we're all adults," presuming said adults cannot be trusted to indent their own code.

Unfortunately, it is difficult to write good object oriented code, or even have an API, with these strictures. We've thus decided to adopt the following convention:

Public

All properties that do not fall into the aforementioned categories are public, i.e., they can be used in any context importing OpenLane.

Public methods are part of the OpenLane API and they are guaranteed to be functional within the same major version.

Protected

Protected methods are marked with the `@protected` decorator. They may be used inside the specific class they are declared in and any subclasses. The `protected`

decorator will append the docstring to clarify that fact.

> **Note**
>
> Some Python conventions specify `_` (a single underscore) for protected methods and properties- however, this can be confusing, especially because some documentation tools such as Sphinx designate all properties and methods starting with `_` as private.

Protected methods are part of the OpenLane API and they are guaranteed to be functional within the same major version.

Internal

Internal properties and methods are prefixed by `_` (one underscore.) They may only be used inside the OpenLane codebase proper and not plugins or the like, even those that inherit from the same classes.

Internal properties and methods are **not** part of the OpenLane API and may break at any time without a major version increment.

Private

Private properties and methods are prefixed by `__` (two underscores.) They may be only be used inside the specific class they are declared in, and not its super or subclasses.

Private properties and methods are **not** part of the OpenLane API and may break at any time without a major version increment.

Hierarchy and “Virtual” Public Variables/Methods

Classes in OpenLane rely on heavy use of polymorphism to define interface by which multiple classes can interact with each other.

By our immutability standard, setters are by definition not in consideration for this codebase. The choice of whether to use a getter or a variable, however, is more involved and objects the following taxonomy:

digraph taxonomy { start [label="Is the value expected to remain constant across all instances of a class?"] class_dependent [label="Is any other class variables or getters?"] class_method [label="Class Method", peripheries=2] class_property [label="Class Property", peripheries=2] start ->

class_dependent [label="Yes"] class_dependent -> class_method [label="Yes"] class_dependent -> class_property [label="No"] dynamic [label="Is the value computed from any other instance variables or getters?"] method [label="Instance Method", peripheries=2] property [label="Instance Property", peripheries=2] start -> dynamic [label="No"] dynamic -> method [label="Yes"] dynamic -> property [label="No"] }

Furthermore, if the method is not useful in a base class, the method (whether for class or instance) must be declared abstract using the `@abstractmethod` decorator. Similarly, properties must be assigned `NotImplemented` to declare that they are abstract. The former is programmatically enforced, but the latter is currently not due to technical limitations.