Writing Custom Steps

Just like you may write custom flows, you may also write custom steps to run within those flows.

Again, please review OpenLane's high-level architecture at this link. This defines many of the terms used and enumerates strictures mentioned in this document.

Generic Steps

Like flows, each Step subclass must:

- Implement the `openlane.steps.Step.run()` method.
  - This step is responsible for the core logic of the step, which is arbitrary.
  - This method must return two values:
    - A `ViewsUpdate`, a dictionary from `DesignFormat` objects to Paths for all views altered.
    - A `MetricsUpdate`, a dictionary with valid JSON values.

> **Important**
>
> Do NOT call the `run` method of any `Step` from outside of `Step` and its subclasses- consider it a protected method. `start` is class-independent and does some incredibly important processing.
>
> You should not be overriding `start` either, which is marked **final**.

But also, each Step is required to:

- Declare any required `openlane.state.State` inputs in the `inputs` attribute.
  - This will enforce checking the input states for these views.
- Declare any potential state modifications in the `outputs` attribute.
  - This list is checked for completeness and validity- i.e. the `Step` superclass WILL throw a `StepException` if a Step modifies any State variable it does not declare.
- Declare any used configuration variables in the `config_vars` attribute.

> **Important**
>
> Don't forget the `Step` strictures. Some of them are programmatically enforced, but some are still not.

Writing Config Variables

Config variables are declared using the `openlane.config.Variable` object.

There are some conventions to writing these variables.

- Variable names are declared in `UPPER_SNAKE_CASE`, and must be valid identifiers in the Python programming language.
- Composite types should be declared using the `typing` module, i.e., for a list of strings, try `typing.List[str]` instead of `list[str]` or just `list`.
  - `list[str]` is incompatible with Python 3.8.
  - `list` does not give OpenLane adequate information to validate the child variables.
- Variables that capture a physical quantity, such as time, distance or similar, must declare units using their `"units"` field.
  - In case of micro-, the only SI prefix denoted with a non-Latin letter, use this exact Unicode codepoint: `µ`
- Variables may be declared as `pdk`, which determines the compatibility of a PDK with your step. If you use a PDK that does not declare one of your declared PDK variables, the configuration will not compile and the step will raise a `openlane.steps.StepException`.
  - PDK variables should generally avoid having default values other than `None`. An exception is when a quantity may be defined by some PDKs, but needs a fallback value for others.
- No complex defaults. Defaults must be scalar and quick to evaluate- if your default value depends on the default value of another variable, for example,
- All filesystem paths must be declared as `openlane.common.Path`, objects which adds some very necessary validation and enables easier processing of the variables down the line.
  - Avoid pointing to entire folders. If your step may require multiple files within a folder, try using the type `List[Path]`.

Implementing `run`

The run header should look like this:

```
def run(self, state_in: State, *args, **kwargs):
```

The `*args` and `**kwargs` allow subclasses to pass arguments to subprocesses- more on that later.

You can access configuration variables- which are validated by this point- using `self.config[KEY]`. If you need to save files, you can get the step directory using `self.step_dir`. For example:

```
design_name = self.config["DESIGN_NAME"]
output_path = os.path.join(self.step_dir, f"{design_name}.def")
```

Note

A step has access to:

- Its declared `config_vars`
- All Common Flow Variables

Attempting to access any other variable is undefined behavior.

Warning

Ensure that, if your configuration variable is **Optional**, that you explicitly check if the variable `is not None`. If the variable is not Optional, validation will handle this check for you.

Otherwise, you're basically free to write any logic you desire, with one exception:

- If you're running a terminal subprocess you'd like to have OpenLane manage the logs for, please use `openlane.steps.Step.run_subprocess()`, passing *args and **kwargs. It will manage I/O for the process, and allow the creation of report files straight from the logs- more on that later.

In the end, add any views updated to the first dictionary in the returned tuple, and any metrics updated to the second dictionary in the returned tuple.

Creating Reports

You can create report files manually in Python, but if you're running a subprocess, you can also write `%OL_CREATE_REPORT <name>.rpt` to stdout and everything until `%OL_END_REPORT` (or another `%OL_CREATE_REPORT`) will be forwarded to a file called `<name>.rpt` in the step dir automatically.

Creating Metrics

Likewise, if you're running a subprocess, you can have `openlane.steps.Step.run_subprocess()` capture them for you automatically by using `%OL_METRIC`. See the documentation of `openlane.steps.Step.run_subprocess()` for more info.

> **Note**
>
> Metrics generated using this method will not be automatically added to the output state. The `openlane.steps.Step.run()` method is expected to capture the returned dictionary of any `openlane.steps.Step.run_subprocess()` invocations and add any values to the returned `MetricUpdate` dictionary as appropriate.

Tool-Specific Steps

The `Step` object makes heavy use of object-oriented programming to encourage as much code reuse as possible. To that extent, there exists some more specialized `Step` abstract base classes that deal with specific utilities:

`openlane.steps.TclStep`

`TclStep` implements a `run` that works for most Tcl-based utilities. This run calls a subprocess with the value of `openlane.steps.TclStep.get_command()`, and it emplaces all configuration variables as environment variables using this scheme:

- List variables are joined with a space character.
- Enumerations are replaced with the enumeration name.
- Booleans are replaced with `"1"` if true or `"0"` if false.
- Integers and Decimals are turned into Base-10 strings.

The state is also exposed to the TclStep as is:

- Input files are pointed to in variables with the format `CURRENT_<view name>`.
- Output paths are pointed to in the variables with the format `SAVE_<view name>`.

If a TclStep-based step fails, a reproducible is created, which can be submitted to the respective repository of the tool.

Keep in mind that TclStep-based tools still have to define their `config_vars`, `inputs` and `outputs`.

Subclasses

`TclStep` has various subclasses for a number of Tcl-based utilities:

- `openlane.steps.OpenROADStep`
- `openlane.steps.YosysStep`
- `openlane.steps.MagicStep`

These subclasses acts as an abstract base class for steps that use their respective utility. They have one abstract method, `get_script_path`. Most steps subclassing them might not need to even override `run`.

Additionally, they comes with a common set of `config_vars` required by all invocations of said tool; you can declare more for your step, however, as shown in this example.:

```
config_vars = OpenROADStep.config_vars + [
    ...
]
```

Be sure to read the subclasses' `run` docstrings as they may contain critical information.