picard Documentation

Release 0.1.3

John Freeman

Dec 29, 2018

Contents

"Make it so." – Captain Jean-Luc Picard

The idea of Ansible with the execution of Make.

With Picard, you define a set of targets, each with a recipe that leaves it in a desired state, e.g. a compiled executable or a running service. Targets may depend on each other, e.g. "this executable depends on that source file" or "this service depends on that host", in a directed acyclic graph. Like Make, Picard executes the recipes for targets in dependency order.

Like Ansible, Picard comes with many sophisticated recipes out-of-the-box that behave like rsync: they find the differences between a target's present state and its goal state, and execute just the changes necessary to transition from the first to the second.

Make is limited to considering targets on the local filesystem, while Ansible can consider more general targets and states, e.g. the existence and configuration of remote machines. Ansible's input is a rigid declarative template (based on Jinja), while Make's input is an executable script that builds the abstract definitions of the targets and gets to leverage functions and variables. Picard tries to combine the best of both worlds in pure Python.

CHAPTER 1

Concepts

Most of the terminology in Picard has been lifted from Make, which I anticipate will be familiar to most of the people reading this. By reusing the same terms for the same or similar concepts, I hope to ease their comprehension.

1.1 Targets

A **target** represents some entity, e.g. a file or a process. It has:

- A unique human-readable name for debugging.
- A (possibly empty) set of **prerequisite** targets.
- A **recipe** function that brings the entity into a specific state, and then returns a representation of it to be used by dependent targets. Recipes are asynchronous to enable concurrency.

```
class Target:
  name: str
  prereqs: Traversable['Target']
  async def recipe(self, context: Context) -> Any: ...
```

In Make, the combination of a target, prerequisites, and a recipe forms a **rule**. We could have called this class a `Rule`, but when we pass these objects around, e.g. as prerequisites, we just think of them as targets, thus the name.

1.1.1 Prerequisites

The type of the prerequisites is a non-specific traversable that supports iteration and mapping. Trivially, this can be a single value, but more often it will be an arbitrary structure of nested collections, like a JSON value. This way, you can use whatever structure you want to express your prerequisites, as long as it can be iterated (to capture the dependency edges) and mapped (for evaluating targets buried within).

1.1.2 Recipes

The basic principle of every recipe is that it establishes a post-condition by the time it exits. In the style of Make, such a post-condition would be "the target file exists with a modified timestamp after that of all of its prerequisites." In the style of Ansible, a post-condition might be "the target service is running on its prerequisite host."

There is one notable difference between Picard's recipes and those of Make. In Make, a recipe is executed conditionally: only when the target does not exist, or has a modified timestamp before one of its prerequisites. In Picard, recipe functions are called unconditionally. The common, expected practice is that recipes themselves check what changes they need to make, and that they make the fewest changes necessary to establish their post-condition. Handing over this responsibility to users is the only way to enable tests beyond modified timestamps.

Each recipe returns an abstract representation of its target, e.g. a file path or a hostname and port. When a target serves as a prerequisite for other targets, its representation may be used in their recipes. When deciding what a recipe should return, consider what dependents may need. It is expected that a recipe returns the same value whether it needed to make changes or not. Recipes should generally memoize their return value to avoid duplicate work.

Each recipe is given an argument called the **context**. Context makes it possible to pass information "up" the dependency graph (or "down", depending on your perspective), from targets to their prerequisites. It generally carries a configuration and a logger.

The process of calling a target's recipe with a context is called **synchronization** or **evaluation**. We generally use "synchronization" to emphasize the process of changing an external entity to match the parameters of the target, and "evaluation" to emphasize the abstract value returned by the recipe in preparation for another recipe.

1.2 Rules

A target is a combination of a name, a set of prerequisites, and an (async) recipe function. Because every async function in Python has a name, we just need to add a set of prerequisites to make a target, which we can do with a decorator much easier than defining a class. Picard calls this decorator `rule` because it lets us build targets from recipe functions with a syntax that mimics Make:

```
@picard.rule()
async def clean(self, context):
    picard.sh('rm', '-rf', 'build')
```

The arguments to the decorator, if any, are the target's prerequisites. The decorated function is its recipe. The first argument to the recipe is the context. The rest of the positional and keyword arguments are the same as what was passed to the decorator, except all targets within will have been replaced by their evaluation.

```
@picard.rule()
async def a(context):
    ...

@picard.rule()
async def b(context):
    ...

@picard.rule(xs=[a, b])
async def target(context, *, xs):
    a, b = xs
    # In here, ``a`` and ``b`` are the values returned by
    # evaluating the targets ``a`` and ``b`` with ``context``.
```

1.3 Patterns

A **pattern** is a template for targets, named after Make's pattern rules. A pattern is first defined by supplying a generic recipe, and then it is instantiated one or more times to make targets.

The recipe given to a pattern definition is much the same as that given to a rule definition, except that it has an additional first parameter: the target itself. A pattern does not yet define a target, so the recipe cannot know it until it is called.

Defining a pattern creates a **constructor** which you can use to stamp out targets. The constructor expects slightly different arguments than the recipe you supplied for the pattern. Its first parameter is the name of the target. The rest of the positional and keyword arguments can be whatever you want to pass through to the recipe. It may contain a mix of values and targets. Any targets nestled within will be considered prerequisites and evaluated before being passed to the recipe. In other words, the recipe will only see values, not targets.

```
import picard

@picard.pattern()
async def object_file(target, context, source):
    await picard.sh('gcc', '-c', source, '-o', target.name)

hello_o = object_file('hello.o', 'hello.c')
example_o = object_file('example.o', source='example.c')
```

1.4 Drivers

Once you've defined a set of rules, you need to choose one or more targets and synchronize them (which will recursively synchronize their prerequisites). Picard offers two functions to help with this.

1.4.1 sync

```
sync(target: Target, context: Context = None) -> Any
```

Synchronize a target with an optional context and return its value. If no context is given, a default context will be constructed, which will have two properties: an empty configuration named `config`, and a logger (the root logger from the `logging` module) named `log`.

1.4.2 make

```
make(
    target: Target,
    config: Mapping[str, Any] = {},
    rules: Mapping[str, Target] = None,
)
```

A command line interface similar to *Make*. `make` takes a few parameters:

1. `target`: The default target to synchronize. In Make, this would be the first declared target. With Picard, you must pass it.
2. `config`: The default configuration, a mapping from strings to values.

3. `rules`: The set of known rules. If not given, it will default to the set of variables in the module from which `make` was called.

`make` takes a few steps:

1. It parses the command line for options of the form `name=value` or `--name value`, and then considers the rest of the command line arguments, if any, to be names of targets.
2. It builds a configuration mapping by taking the defaults in `config`, then overlaying variables from the environment, and then overlaying the options it parsed in step 1.
3. It packages the configuration it built in step 2 with the root logger from the `logging` module into a context.
4. It searches the `rules` mapping for the targets named in step 1 (or if none were found, the default `target`), and then synchronizes them all with the context built in step 3.

`make` is meant to be used like this:

```
import picard

# Define targets.
target = ...

if __name__ == '__main__':
    picard.make(target)
```

CHAPTER 2

API

2.1 Targets

picard.**rule**(**args*, ***kwargs*) → Callable[Callable[[picard.context.Context, Any], Awaitable[Any]], picard.typing.Target]

Turn a recipe function into a target.

The parameters are the prerequisites, which will be passed, evaluated, to the recipe.

Example

```
from pathlib import Path
import picard

@picard.rule()
async def gitdir(context):
    path = Path('.git')
    if not path.is_dir():
        picard.sh('git', 'init', '.')
    return path
```

picard.**pattern**() → Callable[Any, Callable[..., picard.typing.Target]]

Turn a recipe function into a target constructor.

The constructor's parameters are the prerequisites, which will be passed, evaluated, to the recipe.

Example

```
import picard

@picard.pattern()
```

(continues on next page)

(continued from previous page)

```
async def object_file(target, context, source):
    await picard.sh('gcc', '-c', source, '-o', target.name)

hello_o = object_file('hello.o', 'hello.c')
example_o = object_file('example.o', source='example.c')
```

2.2 Drivers

picard.**sync**(*target: Any*, *context: picard.context.Context = None*)

Swiss-army function to synchronize one or more targets.

Parameters

- **targets** – One or more targets. This function will recurse into functors like sequences and mappings. (Remember that mappings are functors over their values, not their keys.)
- **context** – An optional context. If `None` is passed (the default), an empty one will be created for you.

Returns The value(s) of the targets in the same functor.

Return type Traversable[Any]

picard.**make**(*target: Any*, *config: Mapping[str, Any] = None*, *rules: Mapping[str, picard.typing.Target] = None*)

Parse targets and configuration from the command line.

CHAPTER 3

Batteries

They're included.

3.1 Files

Targets based on files and modified timestamps as in Make. File targets evaluate to the `pathlib.Path` of their target.

Consider a simple Makefile that builds an executable from a C source file:

```
hello : hello.o
        $(CC) $(LDFLAGS) -o hello hello.o $(LDLIBS)

hello.o : hello.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -c hello.c
```

Here is the same example in Picard using `picard.file()`. One notable difference is that, due to Python scope rules, you must declare prerequisites before the targets that depend on them.

```
import picard

@picard.file('hello.o', 'hello.c')
async def hello_o(target, context, hello_c):
    cc = context.config.get('CC', 'cc')
    cpp_flags = context.config.get('CPPFLAGS', None)
    c_flags = context.config.get('CFLAGS', None)
    await picard.sh(cc, cpp_flags, c_flags, '-c', hello_c)

@picard.file('hello', hello_o)
async def hello(target, context, hello_o):
    cc = context.config.get('CC', 'cc')
    ld_flags = context.config.get('LDFLAGS', None)
    ld_libs = context.config.get('LDLIBS', None)
```

(continues on next page)

(continued from previous page)

```
    await picard.sh(cc, ld_flags, '-o', 'hello', hello_o, ld_libs)

if __name__ == '__main__':
    picard.make(hello)
```

Note: If you are compiling C or C++, these patterns in this example have already been encapsulated in *picard.clang*.

3.2 C/C++

Patterns for compiling C and C++ objects and executables.

picard.clang.**executable**(*filename: Union[str, os.PathLike], *objects*) → picard.typing.Target
Link an executable from object files.

Parameters

- **filename** – The filename or path to where the executable should be built.
- ***objects** – A set of filenames, paths, or targets for the object files.

Returns A *file target* for the executable linked from `objects`.

Return type Target

picard.clang.**object_**(*source: Union[picard.typing.Target, str, os.PathLike]*) → picard.typing.Target
Compile an object file from a source file.

Parameters **source** – A filename, path, or file target.

Returns A *file target* for the object file compiled from `source`.

Return type Target

picard.clang.**objects**(**sources*)
Return a set of object files mapped from a set of source files.

3.3 Amazon Web Services (AWS)

Patterns for Amazon Web Services resources.

The patterns in this module correspond to AWS resources, e.g. an EC2 instance or an S3 bucket. That is, their post-condition asserts that the resource exists with the given parameters. Each target requires *at least* the parameters necessary to create the resource. These parameters should be enough to identify the resource in a search.

Note: All parameters for these patterns must be keyword arguments.

picard.aws.**security_group**(*name*, **args*, ***kwargs*)
An AWS security group.

Parameters **Description** (*str*) –

picard.aws.**key_pair**(*name*, **args*, ***kwargs*)
An AWS key pair.

Python Module Index

p

Index