Pylightnix

Sergey Mironov

February 2021

Abstract

Pylightnix is a Python domain specific language library for managing filesystem-based immutable data objects, inspired by Purely Functional Software Deployment Model thesis by Eelco Dolstra and the Nix package manager. In contrast to Nix, Pylightnix is primarily focused on managing the data for computer science experiments. Traditional use case of domain-specific package management, as well as other blackboard application use cases are also supported.

With the help of Pylightnix API, applications

- Store the data in form of linked immutable filesystem objects here called **stages**.
- Create (in our terms, **realize**) such objects, access its data, navigate through dependencies.
- Re-run realization algorithms whenever inputs change. Pylightnix may decide to re-create either a whole or a part of the stage object collection according to the changes in prerequisites.
- Manage the outcomes of **non-deterministic** stage realizations. As one example, users may define a Pylightnix stage to depend (in a user-defined sense) on top-10 random instances of a trained machine learning model.

1 Quick start

In this section we illustrate basic principles by defining Pylightnix **stages** required to build GNU Hello program. Resulting code could be used as a part of e.g. a project-specific build system (but rather unsafe one, since Pylightnix lacks build isolation).

1.1 Defining a simple stage

The following code defines a Pylightnix **Stage** entity named `stage_fetch` containing a sources of GNU Hello program. We call a pre-defined `fetchurl` finction which binds some name, URL, and the SHA256 hash string with a built-in algorithm calling well-known `Wget` utility followed by a call to `Aunpack` script.

```
from pylightnix import (Manager, DRef, RRef, fetchurl)

hello_version = '2.10'

def stage_fetch(m:Manager)->DRef: # DRef
  return fetchurl(m,
    name='hello-src',
    url=f'http://ftp.gnu.org/gnu/hello/hello-{hello_version}.tar.gz',
    sha256='31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f51 ⌋
     ↪  6b')
```

- Pylightnix stages are Python functions. Calling a stage function **instantiates** the stage by binding some algorithm with its prerequsites and returns its **derivation references**. Derivation references may be used to introduce dependencies between stages. Including a derivation reference of one stage into a prerequisites of another stage means that the second stage depends on the first one.
- Stage functions take a dependency-resolution **Manager** context as their first arguments. Managers holds the information about instantiated stages. Users don't need to operate on Managers directly. Instead, they are to pass the top-level stage to one of `instantiate`-functions. Returned value would include instantiations of all the stage collection.

1.2 Accessing stage data

We want stage data to appeare in the filesystem so we call `instantiate` function first and then pass its result to `realize` function. Realize function realizes a stage or a sequence of stages and returns the **realization reference** describing the realization outcomes.

Pylightnix API defines the `Lens` 'swiss army knife' for navigating through realization data.

```
 1 from pylightnix import instantiate, realize, mklens
 2 from os.path import join, isdir
 3 
 4 fetch_rref:RRef = realize(instantiate(stage_fetch))  # RRef
 5 print(fetch_rref)
 6 
 7 print(mklens(fetch_rref).val)      # Interpret as Python value
 8 print(mklens(fetch_rref).rref)     # Interpret as RRef (same result)
 9 print(mklens(fetch_rref).dref)     # Interpret as DRef
10 print(mklens(fetch_rref).syspath)  # Interpret as a filesystem path
11 assert isdir(join(mklens(fetch_rref).syspath,
12              f"hello-{hello_version}"))  # SRC
13 print(mklens(fetch_rref).name.val) # Access prerequisite 'name' variable
14                                    # and interpret it as a Python value
15 print(mklens(fetch_rref).url.val)  # Same for 'url' variable
16 print(mklens(fetch_rref).sha256.val)  # Same for 'sha256' variable
```

Output:

```
rref:539eb91fe09f4aac3e1ff4af38c235b5-2f56e6f987a1da0271915894ca19e28f-hello-src
rref:539eb91fe09f4aac3e1ff4af38c235b5-2f56e6f987a1da0271915894ca19e28f-hello-src
rref:539eb91fe09f4aac3e1ff4af38c235b5-2f56e6f987a1da0271915894ca19e28f-hello-src
dref:2f56e6f987a1da0271915894ca19e28f-hello-src
/home/grwlf/_pylightnix/store-v0/2f56e6f987a1da0271915894ca19e28f-hello-src/539eb91fe09f4 ⌋
↪  aac3e1ff4af38c235b5
hello-src
http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f516b
```

- At line 4 we `instantiate` and `realize` our custom stage to get its realization reference.
- Realization is a hash-containing identifier string which could be interpreted as a local filesystem path to a folder containing stage artifacts. The main artifact of `stage_fetch` is the unpacked folder containing the sources of GNU Hello application. At line 12 we check that this folder does indeed exist.

1.3 Defining custom stage

We define a custom Pylightnix stage named `stage_build` to configure and build the GNU Hello application out of its sources.

```
 1 from tempfile import TemporaryDirectory
 2 from shutil import copytree
 3 from os import getcwd, chdir, system
 4 from pylightnix import (Config, Path, Build, mkconfig, mkdrv,
 5   build_wrapper, match_only, dirrw, promise, build_setoutpaths)
 6
 7 def stage_build(m:Manager)->DRef:                               # ST
 8   def _config()->Config:                                        # CFG
 9     name:str = 'hello-bin'
10     fetch_ref:DRef = stage_fetch(m)                             # DEP
11     src:RefPath = [fetch_ref, f'hello-{hello_version}']         # RP1
12     bin:RefPath = [promise, 'usr', 'bin', 'hello']              # RP2
13     return mkconfig(locals())                                   # LOC
14   def _realize(b:Build)->None:                                  # RE
15     build_setoutpaths(b,1)                                      # OP
16     with TemporaryDirectory() as tmp:
17       copytree(mklens(b).src.syspath, join(tmp,'src'))          # LE
18       dirrw(Path(join(tmp,'src')))
19       cwd = getcwd()
20       try:
21         print(f"Building {mklens(b).name.val}")
22         chdir(join(tmp,'src'))
23         system(f'./configure --prefix=/usr')
24         system(f'make')
25         system(f'make install DESTDIR={mklens(b).syspath}')
26       finally:
27         chdir(cwd)
28   return mkdrv(m, config=_config(),
29                   matcher=match_only(),
30                   realizer=build_wrapper(_realize))             # DRV
31
32 build_rref:RRef = realize(instantiate(stage_build))
33 print(build_rref)
```

Output:

```
rref:1e32f4232353722700e2a7c344577766-bee4927b44ba4b7eff25dc7ae142a6f8-hello-bin
```

- At line 7 we define a stage function. Similar to `stage_fetch`, it doesn't take any arguments besides the Manager utility object.
- It is more convenient to start reading the stage code from the end. At line 30 we see the **mkdrv** function call which introduces our stage to Pylightnix. Its three named arguments represent three main components of Pylightnix stages:

  – A **Config** dictionary-like object represents stage's prerequisites. Configs may contain any JSON-serializable fields, that is strings, numbers, booleans, other lists or dicts which match the same rules, etc. Derivation references are represented with Python strings and thus could also be included.
  – A **Matcher** lets Pylightnix decide whether an existing stage object matches its Config specification. As a result the library may either run stage's realizer or re-use existing object or objects.
  – A **Realizer** function knows how to create a new stage object out of its Config prerequisites.

  In this example we expect exactly one outcome so we use a **match_only** matcher which either returns the only existing stage object or asks Pylightnix for a realization.

- At line 8 we define a dictionary containing stage's build prerequisites. For convenience we user a Python trick: define fields as a variables of a `_config` function and then use **locals()** to collect them into a dict. Pylightnix remembers stage prerequisites. Changing any of them would trigger rebuilding.
- At lines 11 and 12 we define **RefPath**s lists which both represent relative paths in the filesystem. A first item of a RefPath should be either a **DRef** derivation reference or a **promise** marker. A promise will be resolved to a future realization path of the stage that makes it.
- By including derivation reference to `stage_fetch` into the config at line 11, we introduce a dependency from `stage_build` to `stage_fetch`! The dependency would make Pylightnix notice changes in `stage_fetch` and rebuild all the dependent stage as expected.
- `mklens` can navigate through dependency reference fields transparently. The definition at line 10 allows us to write:

```
print(mklens(build_rref).fetch_ref.rref)
print(mklens(build_rref).fetch_ref.url.val)
```

  Output:

```
rref:539eb91fe09f4aac3e1ff4af38c235b5-2f56e6f987a1da0271915894ca19e28f-hello-src
http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

1.4 Execute

Finally, we locate the GNU Hello binary and run it.

```python
from subprocess import run, PIPE
print(run([mklens(build_rref).bin.syspath],
↪  stdout=PIPE).stdout.decode('utf-8'))
```

Output:

```
Hello, world!
```