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 Features

We tried to meet high development standards. In particular, Pylightnix:

- Is written in Python 3.7. Mypy typing is used whenever possible.
- Is tested using Hypothesis.
- Documentation is written in a literate programming style. Most of the code examples shown in this manual were checked and evaluated inline by PythonTex tool.
- Core modules depend solely on the Python standard library. Optional modules do depend on Curl and Wget for accessing the Internet and on Atool for dealing with compressed files.
- Alas, Pylightnix is not a production-ready yet! No means of parallelism are provided, network synchronization is yet under development. We didn't check Pylightnix on any operating system besides Linux. We tried our best to make Pylightnix' storage operations atomic. Among other benefits, this allows running several instances of the library on a single storage at once.

2 Install

Pylightnix could be installed either by running Pip package manager as usual or by bundling the package from source. Since we are in deep betas, you probably should prefer source-based installation. We will also need Curl and ATool system packages later in this manual.

2.1 Install with Pip

```
$ pip3 install pylightnix
```

2.2 Build from source

1. Clone the repo

   ```
   $ git clone https://github.com/stagedml/pylightnix
   $ cd pylightnix
   ```

2. Either

   - Setup 'PYTHONPATH' to point to the sources.

     ```
     $ export PYTHONPATH="`pwd`/src:$PYTHONPATH"
     ```

     Now you could import pylightnix from your applications.
   - Build and install pylightnix wheel.

     ```
     $ make wheels
     $ sudo -H pip3 install --force dist/*whl
     ```
   - Nix users may refer to default.nix and shell.nix expressions.

3. (Optional) Run the tests.
4. (Optional) Make docs
5. (Optional) Build the demos

2.3 Install recommended system packages

Pylightnix utilities rely on Curl and ATool system packages. Installing them is highly advised.

```
$ apt-get install -y curl atool       # Use your system's package manager here!
...
$ curl --version | head -n1
curl 7.70.0
$ aunpack --version | head -n1
atool 0.39.0
```

2.4 Make sure GNU Autotools are available

We will also need GNU Autotools for the sake of demonstration. Make sure they are installed in the system.

3 Related work

- Nix (Nix repo, Comparison)
- Spack
- Popper
- CK

4 Quick start

We illustrate Pylightnix principles by showing how to use it as a Nix-like build system. We will define two **stages** required to build GNU Hello application.

Note that despite using similar approaches, Pylightnix doesn't aim to be a direct Nix replacement. It lacks many features you would probably expect of a OS-level package management tool. It has no multiprocessing support neither it has a build isolation. In contrast, Pylightnix is a bare-bone library you could easily integrate into packages to handle project-level data management problems.

4.1 Defining a Stage

The following Python function defines a Pylightnix **Stage** entity named `stage_fetch`. In its body we use a pre-defined `fetchurl` finction which binds some name, URL, and the SHA256 hash string with a built-in algorithm calling `Wget` and `Aunpack` tools to download GNU Hello sources from the Internet.

```
1  from pylightnix import (Manager, DRef, RRef, fetchurl)
2
3  hello_version = '2.10'
4
5  def stage_fetch(m:Manager)->DRef: # DRef
6    return fetchurl(m,
7      name='hello-src',
8      url=f'http://ftp.gnu.org/gnu/hello/hello-{hello_version}.tar.gz',
9      sha256='31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f51
       ↪  6b')
```

- Indeed, Pylightnix stages are Python functions. By calling a stage function we register or **instantiate** this stage, that is, we bind some realization algorithm with its input arguments here called prerequisites.
- Stage function takes a dependency-resolution **Manager** context as its first arguments. The Manager holds the information about instantiated stages. Users normally shouldn't operate on Managers or call stage functions directly. Instead, they are to pass the top-level stage to one of Pylightnix top-level `instantiate-` functions.
- The return value of a stage function is a **derivation reference**. The primary role of derivation references is to introduce new dependencies between stages. By including a derivation reference of one stage into prerequisites of another stage we say that the second stage depends on the first stage.

4.2 Accessing the Data

The primary method of accessing stage data is to call `realize` function, passing the `instantiated` stage as its argument. Pylightnix either runs the algorithm associated with a stage or re-uses existing realization. In both cases, a **realization reference** handle is returned. We then use a `Lens` helper which is a 'swiss army knife' for navigating through the Pylihgtnix storage.

```
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 reference is a string identifier which could be converted to a local filesystem path within the storage. This path contains the sources of GNU Hello application. At line 12 we check that this path is valid.

4.3 Defining a Custom Stage

In this section we define a custom Pylightnix stage named `stage_build`. In its algoritm we provide instructions how to 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.
- It is more convenient to start reading the stage code from the end. At line 30 we call the **mkdrv** stage constructor. The function has the following arguments:
  - **Manager** utility object. Just pass the Manager we get from the stage function arguments.

  - **Config** should be a dictionary-like object representing stage's prerequisites. Configs may only contain JSON-serializable fields, that is strings, numbers, booleans, lists or other dictionaries.
  - **Matcher** function lets Pylightnix decide whether to run realizer or to re-use an existing object or objects.
  - **Realizer** function knows how to create a new stage object out of prerequisites.

  In this example we expect to get exactly one outcome. We use a simple **match_only** matcher which ensures that our stage has only one realization.

- At line 8 we define our stage's build prerequisites. For convenience we user a Python lifehack - call `local()` to convert local variables of `_config()` into a dictionary.
- At line 11 we define a **RefPath**, a path, relative to a stage realization folder. This particular path refers to a GNU hello source folder which we expect to see among `stage_fetch`'s data. `mklens` knows how to convert such paths into a regulat system paths.
- At line 12 we also define a **Promise Path**. Pylightnix will replace promise markers with a derivation reference of the stage being built and checks that said object does really exist in the filesystem.
- By mentioning derivation `fetch_ref` at line 10, we say that `stage_build` depends on `stage_fetch`. Later we use this dependency at line 17 to access the `stage_fetch`'s output data.
- With `mklens` we could explore dependencies in-depth. The definition at line 10 allows us to read the parent's prerequisites as follows:

```
1  print(mklens(build_rref).fetch_ref.rref)
2  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
```

4.4 Execute

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

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

Output:

```
Hello, world!
```

5 Rationale

5.1 Why Nix-based?

There are many solutions in the area of software deployment. Besides Nix, we know all the traditional package managers, Docker, AppImage, VirtualBox and so on. One property of Nix we want to highlight is it's low system requirements. Basically, Nix core needs only a basic file system API in order to work. Here we try to follow the trend of keeping the number of dependencies, and still provide a competitive set of features.

5.2 Why functional-style API?

There are several reasones:

- We think that this way we could track the API changes more easier. We are trying to avoid changes in functions which are already published. We tend to import functions by name to let Python notify us whenever the API is changed.
- Class-based APIs of Python often mislead users into thinking that they could extend it by sub-classing. We don't support extention by-subclassing and so we don't need classes.
- Class-based API wrappers may be created as a standalone module. A one example of such a wrapper is the **Lens** module.