Pylightnix

Sergey Mironov

February 2021

Abstract

Pylightnix is a minimalistic Python library for managing filesystem-based immutable data objects, inspired by Purely Functional Software Deployment Model thesis by Eelco Dolstra and the Nix package manager.

The library may be thought as of low-level API for creating caching wrappers for a subset of Python functions. In particular, Pylightnix allows user to

- Prepare (or, in our terms, **instantiate**) the computation plan aimed at creating a tree of linked immutable stage objects, stored in the filesystem.
- Implement (**realize**) the prepared computation plan, access the resulting artifacts. Pylightnix is able to handle possible **non-deterministic** results of the computation. As one example, it is possible to define a stage depending on best top-10 instances (in a user-defined sence) of prerequisite stages.
- Handle changes in the computation plan, re-use the existing artifacts whenever possible.
- Gain full control over all aspects of the cached data including the garbage-collection.

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 to deal 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

2.1 Install with Pip

```
$ pip3 install pylightnix
```

Note that Pylightnix is currenlty under development. PyPi database may contain an outdated version.

2.2 Build from source

1. Clone the repo

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

   As a suggestion, consider making Pylightnix a part of a larger project. In Git VCS, one would typically use `git submodule add` to link Pylightnix to a project as a submodule.

2. Either:

   - Set the `PYTHONPATH` environment variable to import Pylightnix in-place. Currently we recommend to prefer this way of importing the library.

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

     MyPy type checker may also requre a setting of `MYPYPATH` environment variable: `export MYPYPATH=‘pwd‘/src:‘pwd‘/tests`

   - Build and install the wheel package.

     ```
     $ make wheel
     $ sudo -H make install
     ```

See the Development section of the Manual for a detailed guidance on operating a development environment.

2.3 Install the 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 GCC and GNU Autotools are available

Some demonstration code from this manual also uses GCC and GNU Autotools. Make sure they are installed.

3 Related work

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

4 Quick start

We illustrate Pylightnix principles by showing how to use it for making a mock data science task. We'll follow the SciPy anneal example but imagine that we want to save some intermediate results and also that there are several people working on this task.

We start by importing the required Python modules.

```
import numpy as np
import scipy.optimize as o
import matplotlib.pyplot as plt

from pylightnix import *
```

4.1 The problem

The annealing example begins with defining a complex parametric function `f` for what we need to find a minimum point as close as possible.

```
def f(z, *params):
    x, y = z
    a, b, c, d, e, f, g, h, i, j, k, l, scale = params
    return (a * x**2 + b * x * y + c * y**2 + d*x + e*y + f) + \
           (-g*np.exp(-((x-h)**2 + (y-i)**2) / scale)) + \
           (-j*np.exp(-((x-k)**2 + (y-l)**2) / scale))
```

4.2 Stages basics

Lets assume that we are to get the `f`'s parameters from elsewhere and that we want to get the following results: (a) the starting parameters themselves (b) the annealing result and statistics and (c) visual plot of the results.

4.2.1 Parameter stage

The Pylightnix Stage is a Python function that registers a user-defined action in the Manager and returns a DRef reference stating that the action is accepted.

```
def stage_params(m:Manager)->DRef:
  def _config():
    name = 'params'
    out = [selfref, "params.npy"]
```

```
    return locals()
  def _make(b:Build):
    np.save(mklens(b).out.syspath, (2, 3, 7, 8, 9, 10, 44, -1, 2, 26, 1,
    ↪  -2, 0.5))
  return mkdrv(m, mkconfig(_config()), match_only(), build_wrapper(_make))
```

The actual registration is done by the mkdrv function which accepts the manager and the following three arguments (a) the action Config object; (b) the Matcher callback function; (c) the action callback accepting the Build task description. In this example we are pretending to receive input parameters from a thirdparty. We could just as well have downloaded them from the Internet using the fetchurl pre-defined stage.

A few notes on the above code:

- We define configuraiton dictionary by reading a local variables of a helper function named `_config()`. The resulting dictionary should match the Config requirements. As often in Pylightnix, we use a standalone mkconfig construcor rather than create the object directly.
- By specifying a `selfref` output path we promise to produce an artifact with this name. Pylightnix will throw an error if it does not find such an artifact after the build action completes.
- When inside the build action, we refer to the pylightnix config items by using the mklens swiss knife helper. `mklens` returns the Lens object which has an overloaded dot `"."` operator. The middle part of the lens expression encodes the path in the configuration and the last word `"syspath"` encodes the format of the result.
- We pass a match_only() matcher in the third argument of `mkdrv`. By this we expect our action to return a determenistic result. Pylightnix will throw an error if it is not the case.

4.2.2 Annealing stage

```
def stage_anneal(m:Manager, ref_params:DRef)->DRef:
  def _config():
    name = 'anneal'
    nonlocal ref_params
    trace_xs = [selfref, 'tracex.npy']
    trace_fs = [selfref, 'tracef.npy']
    out = [selfref, 'result.npy']
    return locals()
  def _make(b:Build):
    params = np.load(mklens(b).ref_params.out.syspath)
    xs = []; fs = []
    def _trace(x,f,ctx):
      nonlocal xs,fs
      xs.append(x.tolist())
      fs.append(f)
    res = o.dual_annealing(f, [[-10,10],[-10,10]],
                           x0=[2.,2.],args=params,
                           maxiter=500, callback=_trace)
```

```
      np.save(mklens(b).trace_xs.syspath, np.array(xs))
      np.save(mklens(b).trace_fs.syspath, np.array(fs))
      np.save(mklens(b).out.syspath, res['x'])
    return mkdrv(m, mkconfig(_config()), match_only(), build_wrapper(_make))
```

4.2.3 Plotting stage

```
def stage_plot(m:Manager, ref_anneal:DRef)->DRef:
  def _config():
    name = 'plot'
    nonlocal ref_anneal
    out = [selfref, 'plot.png']
    return locals()
  def _make(b:Build):
    xs=np.load(mklens(b).ref_anneal.trace_xs.syspath)
    fs=np.load(mklens(b).ref_anneal.trace_fs.syspath)
    res=np.load(mklens(b).ref_anneal.out.syspath)
    plt.figure()
    plt.title(f"Min {fs[-1]}, found at {res}")
    plt.plot(range(len(fs)),fs)
    plt.grid(True)
    plt.savefig(mklens(b).out.syspath)
  return mkdrv(m, mkconfig(_config()), match_latest(),
   ↪  build_wrapper(_make))
```

4.2.4 Running the experiment

```
ds=instantiate_inplace(stage_params)
cl=instantiate_inplace(stage_anneal,ds)
vis=instantiate_inplace(stage_plot,cl)
rref=realize_inplace(vis)
print(rref)
```

```
rref:cb252162ae8a0fe2c90fe48193f851f8-ff1dd5c155ee3faa445572e3246f0566-plot
```

4.3 Nested stages

In the previous sections we implicitly used a global Manager to register our stages. A more clean approach requires us to define a nested stage covering the whole experiment. This way we pospone the problem of choosing which Manager to use.

```
def stage_all(m:Manager):
  ds=stage_params(m)
  cl=stage_anneal(m,ds)
  vis=stage_plot(m,cl)
  return vis
```

The nested stage may be instantiated and realized in one go.

```
rref_all=realize(instantiate(stage_all))
assert rref_all==rref
print(rref_all)
```

```
rref:cb252162ae8a0fe2c90fe48193f851f8-ff1dd5c155ee3faa445572e3246f0566-plot
```

5 Core entities

Artifacts is any object we could store in the filesystem. Typically, it is either a file or a folder which has a name and an arbitrary content.

6 Pylightnix for Nix users

- Like Nix, Pylightnix offers purely-functional solution for data deployment problem.
- Like Nix, Pylightnix allows user to describe and run two-phased build processes in a controllable and reproducible manner.
- Unlike Nix, pylightnix doesn't aim at providing operating system-wide package manager. Instead it tries to provide a reliable API for application-wide storage for immutable objects, which could be a backbone for some package manager, but also could be used for other tasks, such as data science experiment management.
- Unlike Nix, Pylightnix doesn't provide neither interpeter for separate configuration language, nor build isolation. Both instantiation and realization phases are to be defined in Python. There are certain mechanisms to increase safety, like the recursion checker, but in general, users are to take their responsibility of not breaking the concepts.
- Unlike Nix, Pylightnix is aware of non-deterministic builds, which allows it to cover a potentially larger set of use cases.

Nix

```
{ pkgs? import <nixpkgs> {}
, stdenv ? pkgs.stdenv
}:

let
  version = "2.10";
  url = "mirror://gnu/hello/hello-\${version}.tar.gz";
  sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
in
with pkgs;
stdenv.mkDerivation rec {
  name = "hello";
  src = fetchurl { inherit url sha256; };

  buildInputs = [ gnutar ];

  buildCommand = ''
    tar -xzf $src
    cd hello-2.10
    ./configure --prefix=$out
    make
    make install
  '';
}
```

Pylightnix

```
from tempfile import TemporaryDirectory
from shutil import copytree
from os import getcwd, chdir, system
from os.path import join
from pylightnix import (
  Manager, DRef, RRef, Config, RefPath,
  mkconfig, Path, Build, build_outpath, mkdrv,
  build_wrapper, match_latest, dirrw, mklens,
  instantiate, realize, fetchurl, selfref)

version = '2.10'
url = f'http://ftp.gnu.org/gnu/hello/hello-{version}.tar.gz'
sha256 = '31e066137a962676e89f69d1b65382de95a7ef7d914b8cb956f41ea72e0f516b'

def stage_hello(m:Manager)->DRef:

  def _config()->Config:
    name:str = 'hello-bin'
    src:RefPath = fetchurl(m, name='hello-src',
                           url=url,
                           sha256=sha256)
    out_bin = [selfref, 'usr', 'bin', 'hello']
    return locals()

  def _make(b:Build)->None:
    o:Path = build_outpath(b)
    with TemporaryDirectory() as tmp:
      copytree(join(mklens(b).src.syspath, f'hello-{version}'),
      ↪  join(tmp,'src'))
      dirrw(Path(join(tmp,'src')))
      cwd = getcwd()
      try:
        chdir(join(tmp,'src'))
        system('./configure --prefix=/usr')
        system('make')
        system(f'make install DESTDIR={o}')
      finally:
        chdir(cwd)

  return mkdrv(m, mkconfig(_config()),
                  match_latest(),
                  build_wrapper(_make))
```

7 Rationale

7.1 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.

7.2 Why functional-style API?

There are several reasons:

- 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 extension 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.

7.3 What is the idea behind the promises?

Promises plays a role of a 'type-checker' which notice erroneous realizations before they appear in the Storage just like the regular type-checkers make sure that no erroneous programs appear as binaries.

8 Development

8.1 Environment setup

While Pylightnix enjoys minimum run-time dependencies, it does require lots of development dependencies to be installed. We recomend to install Nix package manager to install all these dependencies automatically. There are following options available:

- File `default.nix` contains the expression for building the end-user Python package. To run the build, execute `nix-build` in the project's root folder:

  ```
  $ nix-build
  ```

  The result will appear under the symbolic link `./result` in the current folder.

- In `shell.nix` we define dependencies and specify configuration of the development shell. Use standard `nix-shell` tool to enter this shell:

  ```
  $ nix-shell
  ```

  Inside the shell, one would use `GNU Make` with the following targets.

  - `make tc` - Typechecks the project using MyPy
  - `make coverage` and `make test` (a synonym) - Run Pylightnix test and prepare the coverage reports.
  - `make docs` - Builds the Pylightnix API reference and Tex documentation.
  - `make wheel` - Builds the wheel pip package
  - `sudo -H make install` - For non-Nix systems: installs the package systemwide. This target should be called after a successful call to `make wheel`. Nix users should import `default.nix` where requred or use `nix-build`.

  The below utilitary targets rely on private account credentials which are not included. One would need to adjust some settings in order to make it work.

  - `make publish-docs` - Commits documentation PDFs to Author's docs repository. Change the repository URL in the Makefile before using this target.
  - `make coverage-upload` Uploads the test coverage report to the Author's CodeCov account. There must be a file `.codecovrc` containing the CodeCov access token.

8.2 References

- Python:
  - Approaches to versioning
  - Setuptool-scm versioning
  - On Manifests. Manifests seem to be useless in our case.
- Tex
  - Link TexOverflow answer about math notation with a nice math-syntax example
  - Link A hint on the Set theory markup: