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 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 *
```

1.1 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))
```

1.2 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.

1.2.1 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.

1.2.2 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))
```

1.2.3 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))
```

1.2.4 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
```

1.3 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
```