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 (**realize1**) the prepared computation plan, access the resulting artifacts. Pylightnix is able to handle **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 sense) 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.

Contents

1 Quick start

We illustrate Pylightnix principles by showing how to use it for making a mock data science task. We will follow the SciPy annealing example but imagine that we additionally 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 problem begins with defining a parametric function `f` for which we need to find an approximate minimum by running the simulated annealing algorithm.

```
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

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

1.2.1 Parameter stage

Pylightnix Stage is a Python function that registers a user-defined action in the Manager (passing None means asking Pylightnix to use the default one) and returns a DRef reference stating that the action is accepted.

```python
def stage_params(m:Optional[Manager]=None)->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(mkconfig(_config()), match_only(), build_wrapper(_make), m)
```

Here, a user-defind action `_make` goes to Pylightnix by calling the mkdrv with the Manager and the following three arguments: (a) the set of prerequisites for the action; (b) the Matcher callback function; (c) the action itself that takes a Build context as an argument. In this example we are pretending to receive input parameters from a third-party. We could just as well have downloaded them from the Internet using a pre-defined stage fetchurl.

A few notes on the above code:

- We define configuration object by reading a local variables of a helper function named `_config()`. The resulting dictionary should match the Config requirements. As is often the case in Pylightnix, we use a standalone mkconfig constructor rather than create the object directly.
- We promise to produce an artifact named `out` by specifying its `selfref` path. 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 function. `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 expression result.
- We pass a match_only() matcher in the third argument of `mkdrv`. By this we say that we expect our action to return a deterministic result. Later we will see how to deal with non-deterministic actions.

1.2.2 Annealing stage

```python
def stage_anneal(ref_params:DRef, m:Optional[Manager]=None)->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(mkconfig(_config()), match_only(), build_wrapper(_make), m)
```

1.2.3 Plotting stage

```
def stage_plot(ref_anneal:DRef, m:Optional[Manager]=None)->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(mkconfig(_config()), match_latest(), build_wrapper(_make),
  ↪   m)
```

1.2.4 Running the experiment

To run the experiment we register all the stages in the internal manager by calling `instantiate` and ask Pylightnix to provide us with the realization reference by calling `realize1`.

```
with current_manager(Manager()):
  ds=stage_params()
  cl=stage_anneal(ds)
  vis=stage_plot(cl)
  rref=realize1(instantiate(vis))
  print(rref)
```

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

1.3 Nested stages

In the previous sections we used a global Manager object to register stages. Another way of combining stages together is to wrap them with an umbrella stage covering the whole experiment.

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

We may pass the top-level stage directly to `instantiate` which would call it with a local disposable Manager.

```
rref_all=realize1(instantiate(stage_all))
assert rref_all==rref
print(rref_all)
```

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

1.4 Collaborative work

Alice and Bob runs the same experiment using their machines. We model this situation by running Pylightnix with different storage settings.

```
Sa=mkSS('_storageA')
Sb=mkSS('_storageB')
fsinit(Sa,remove_existing=True)
fsinit(Sb,remove_existing=True)
rrefA=realize1(instantiate(stage_all, S=Sa))
rrefB=realize1(instantiate(stage_all, S=Sb))
print(rrefA)
print(rrefB)
```

Our annealing problem is stochastic by nature so the results naturally differ.

1.5 Synchronization

Alice sends her results to Bob, so he faces a problem of picking the best realizaion.

```
print("Bob's storage before the sync:")
for rref in allrrefs(S=Sb):
  print(rref)
arch=Path('archive.zip')
spack([rrefA], arch, S=Sa)
# .. Alice transfers the archive to Bob using her favorite pigeon post
↪  service.
sunpack(arch, S=Sb)
print("Bob's storage after the sync:")
for rref in allrrefs(S=Sb):
  print(rref)
```

```
Bob's storage before the sync:
rref:6f906ca9ad8d9b128c57ed0df95ab899-7c308bad0e7fa364790b71e6ffab924a-params
rref:6308158f24f170c4c516275937590fdd-aeeb4bea649e22497b49dfc1ec5ebedf-anneal
rref:81a4e5cbafc86aad2600872d0e109ba8-ff1dd5c155ee3faa445572e3246f0566-plot
Bob's storage after the sync:
rref:6f906ca9ad8d9b128c57ed0df95ab899-7c308bad0e7fa364790b71e6ffab924a-params
rref:c4af92dc7fb53ed854a5d9c8c4c7916c-aeeb4bea649e22497b49dfc1ec5ebedf-anneal
rref:6308158f24f170c4c516275937590fdd-aeeb4bea649e22497b49dfc1ec5ebedf-anneal
rref:81a4e5cbafc86aad2600872d0e109ba8-ff1dd5c155ee3faa445572e3246f0566-plot
rref:b3c26f54e7a81316e567152fbdc1b14a-ff1dd5c155ee3faa445572e3246f0566-plot
```

Bob writes a custom matcher that selects the realization which has the best annealing result.

```python
def match_min(S, rrefs:List[RRef])->List[RRef]:
  avail=[np.load(mklens(rref,S=S).trace_fs.syspath)[-1] for rref in rrefs]
  best=sorted(zip(avail,rrefs))[0]
  if best[1] in allrrefs(Sa):
    print(f"Picking Alice ({best[0]}) out of {avail}")
  else:
    print(f"Picking Bob ({best[0]}) out of {avail}")
  return [best[1]]

def stage_all2(m:Manager):
  ds=stage_params(m)
  cl=redefine(stage_anneal, new_matcher=match_min)(ds,m=m)
  vis=stage_plot(cl,m)
  return vis

rref_best=realize1(instantiate(stage_all2,S=Sb))
print(rref_best)
```

```
Picking Alice (-3.302737056401611) out of [-3.302737056401611, 1.6165660253729486]
rref:b3c26f54e7a81316e567152fbdc1b14a-ff1dd5c155ee3faa445572e3246f0566-plot
```