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))
```

Suppose we are given the parameters for `f` and our engineering goals are: (a) to save the parameters (b) to run the annealing simulation and (c) to make a visual plot of the result.

1.2 Pylightnix stages

Pylightnix offers the Stage mechanism to organize non-deterministic Python functions into a directed acycilc graph, where each node has the following attributes:

- config - a set of user-defined pre-requisite parameters and references to parent nodes.
- matcher - either built-in or user-defined Python function to filter subsets of possible non-deterministic results.

- realizizer - a user-defined function producing one or many sets of possibly non-deterministic results.

1.2.1 Parameter stage

The shortest way of defining Stages is to wrap a future-realizer function with the autostage decorator.

```
@autostage(name='params',
           out=[selfref, "params.npy"])
def stage_params(build:Build, name:str, out:str)->DRef:
  np.save(out, (2, 3, 7, 8, 9, 10, 44, -1, 2, 26, 1, -2, 0.5))
```

The decorator accepts an arbitrary number of parameters which together form the set of pre-requisites for our stage. The decorator passes most of them to the wrapped function, but it changes some types according to the rules explained below and in the decorator's reference.

In the above code the `selfref` list of strings named `out` is automatically translated to a system path with the same name. Upon the realizer's completion, this path should contain a file or a folder. Pylightnix doesn't check its contents in any way, but not having this file at all is an error.

The decorator also adds the build context object named Build (which is not used here) and changes the type of the function result from `None` to DRef which stands for "Derivation Reference". Derivation references may be used to either run corresponding stages or to link them to other dependant stages.

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.

1.2.2 Annealing stage

```
@autostage(name='anneal',
           trace_xs=[selfref, 'tracex.npy'],
           trace_fs=[selfref, 'tracef.npy'],
           out=[selfref, 'result.npy'],
           sourcedeps=[f])
def stage_anneal(build:Build, name,
                 trace_xs:str, trace_fs:str, out:str,
                 ref_params)->DRef:
  params = np.load(ref_params.out)
  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(trace_xs, np.array(xs))
  np.save(trace_fs, np.array(fs))
  np.save(out, res['x'])
```

In the above stage we see another special decorator parameter named `sourcedeps`. This parameter accepts a list of arbitrary Python objects whose source we want Pylightnix to include into the set of pre-requisites. Here we want Pylightnix to re-build the stage every time the `f`'s sources change.

Also note `ref_params` parameter of the realizer which is not included into the decorator. This parameter will be provided additionally as a `DRef` reference. Yes, this is how we specify the dependencies between stages. The decorator translates the reference into a Python object which has all parent pre-requisites visible as attributes. Pylightnix guarantees that all selfref paths of the dependency (like `out` here) do point to valid file-system objects at the time of stage execution.

1.2.3 Plotting stage

```
@autostage(name='plot', out=[selfref, 'plot.png'])
def stage_plot(build:Build, name:str, out:str, ref_anneal:dict)->DRef:
  xs=np.load(ref_anneal.trace_xs)
  fs=np.load(ref_anneal.trace_fs)
  res=np.load(ref_anneal.out)
  plt.figure()
  plt.title(f"Min {fs[-1]}, found at {res}")
  plt.plot(range(len(fs)),fs)
  plt.grid(True)
  plt.savefig(out)
```

1.2.4 Running the experiment

To run the experiment we link stages together by calling them under the control of dependency Registry. We pass the last reference to `instantiate` and `realize1` functions. The realize run the realizares in the right order and returns a second kind of reference called realization reference. This new reference points to both the original derivation and to a particular set of build results.

```
with current_registry(Registry()):
  dref_params = stage_params()
  dref_anneal = stage_anneal(ref_params=dref_params)
  dref_plot = stage_plot(ref_anneal=dref_anneal)
  rref = realize1(instantiate(dref_plot))
  print(rref)

  assert isdref(dref_plot)
  assert isrref(rref)
```

```
rref:c9ccfa58df7d05b0882b67081ac5e8ce-9d94881fd61bba79aa3f83b04ab2d1de-plot
```

1.3 Nested stages

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

```
def stage_all(r:Registry)->DRef:
  ds = stage_params(r)
  cl = stage_anneal(r,ref_params=ds)
  vis = stage_plot(r,ref_anneal=cl)
  return vis
```

As you can see, the umbrella stage is a regular Python function which takes a `Registry` and returns a `DRef`.

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

```
rref_2 = realize1(instantiate(stage_all))
assert rref_2==rref
print(rref_2)
```

```
rref:c9ccfa58df7d05b0882b67081ac5e8ce-9d94881fd61bba79aa3f83b04ab2d1de-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:6f906ca9ad8d9b128c57ed0df95ab899-72f9fb312839bb679f17ab72ac121b81-params
rref:c4ca16b507e657db018ac8d694b75972-ea1827dfb45ee9d15cb1fed42e313041-anneal
rref:0f2caad60ab159fb3d47edbc04371874-9d94881fd61bba79aa3f83b04ab2d1de-plot
Bob's storage after the sync:
rref:6f906ca9ad8d9b128c57ed0df95ab899-72f9fb312839bb679f17ab72ac121b81-params
rref:c4ca16b507e657db018ac8d694b75972-ea1827dfb45ee9d15cb1fed42e313041-anneal
rref:13b9229decd672e798222efd30e3ce2d-ea1827dfb45ee9d15cb1fed42e313041-anneal
rref:0f2caad60ab159fb3d47edbc04371874-9d94881fd61bba79aa3f83b04ab2d1de-plot
rref:376045a6623e3a185fd669acab8dcfe6-9d94881fd61bba79aa3f83b04ab2d1de-plot
```

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

```
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(r:Registry):
  ds=stage_params(r)
  cl=redefine(stage_anneal, new_matcher=match_min)(r=r,ref_params=ds)
  vis=stage_plot(r,ref_anneal=cl)
  return vis

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

```
Picking Alice (-3.408582123530715) out of [-3.302737056408162, -3.408582123530715]
rref:376045a6623e3a185fd669acab8dcfe6-9d94881fd61bba79aa3f83b04ab2d1de-plot
```