Design notes

The pdart codebase was designed as it went, with requirements changing over time. This makes for a somewhat inconsistent design. Some elements of the design were intentional; others were historical accidents. I'll try to document some of them here.

The code is *not* "Pythonic". My background is with strongly typed, compiled languages where the compiler finds errors before the code runs. Python is not such a language. But given the large amount of data we process in this project, running a hour-long program only to have it crash last-minute because of a typo is grossly inefficient. I started using `mypy` when the project was still in Python 2.7, but `mypy` didn't work especially well with it. When we switched to Python 3, `mypy` began to shine and saved me hours and hours of time.

The code uses types extensively to prevent mistakes. There are no cases where you can, for example, give *either* a `pdart.pds4.LIDVID` object *or* a string that represents a LIDVID to the same function, something commonly done in Python. `pdart` functions are rigid in what types they take; I've found this very useful in preventing bugs.

I often use assertions to state my assumptions at the beginning of a function, and at the end of a function to verify that the function did what it was intended to. These

assertions make the code more verbose, but the payoff is that when you break something, the code will often tell you immediately what's wrong and point you to the site where it failed.

Most of the code is written to be automated. I did very little development in a REPL. I'd intended that we could set up a nightly run of the test suite (and I *did* run one at home for a while). For the test suite to work, it needs to be able to be run automatically. So if I could easily capture a property of the code, I'd write a unit test for it.

Unfortunately, much of our data is too complex to create in a test suite. For instance, some functions that process files assume that the file is a legal FITS file, which is too complex an object to create on the fly. So the unit-test suite is, unfortunately, incomplete.

First correct, then fast

Much of the code is inefficient. I don't think speed really matters for us: the files we process don't change very often —perhaps a few times a year—so if the code takes 10 minutes or ten hours doesn't really matter. But whether the code is correct or not *is* important. (After all, if you don't care if your answer is correct or not, I can give you an incorrect answer as fast as you like.)

In the `pdart.fs` (filesystems) code, the `primitives`

module contains a filesystem implementation that's incredibly inefficient, but easy to prove correct. I used that to develop some of the other filesystem implementations. If you look at it and wonder why anyone would write such inefficient code, that's why: to make sure the code was correct.

There are some calculations that make multiple calls to the database, then process the results in Python. It may very well be possible to speed these up with clever use of SQL joins, but it was easy (and provably correct) to code it in Python.

The `pdart.xml` code is written backwards for historical reasons. When it was first written, we were planning to create the XML as we read in data, but the data arrived bottom-up while the standard Python library creates the XML structure top-down. So instead of handling XML directly, we work with *functions that build XML*. Later on, we decided to load the entire contents of the FITS files into the database so, given random access to the data, working bottom-up was no longer necessary.

In the pipeline, I use multiple copy-on-write filesystems when a single one would do. (Look for filesystems with `delta` in their name.) This was done so that if the pipeline was buggy, I could tell which stage had the bugs. It might be cleaner to use a single copy-on-write filesystem.

Any of this code could be rewritten to be more efficient if needed. I'd recommend working through a transition stage where both the original, proveable correct version of the code and the new, optimized version of code both run, and the results are compared to make sure they match.

If you don't need speed, leave it as it is.

Failed or incomplete ideas

logging: I added logging into the code in my last week when it became clear that I needed some way to do a post-mortem on pipeline passes that fail. The basic functionality is in there, but incompletely used.

single-versioned-ness and multiversioned-ness: at some point in building the pipeline, I was juggling five or six filesystems, some of which were dealing with a single version, and others that were dealing with archives containing multiple versions. From the filesystem, it was hard to tell which was which. I thought it would be useful to suffix directory names with `-sv` or `-mv` and to make different functions for the two kinds of filesystems. It helped some, but not a lot, and it was never fully implemented.

the pipeline: The pipeline is a framework and was thrown together to test code. It was never intended to be part of the final product, at least not without rewriting. Think of it as a set of blocks to build the real pipeline.

Random notes:

• The PDS4 format is complex and sometimes almost circular in its requirements. Some things having to do with bundles need to know about collections, and some things having to do with collections need to know about the bundles they're in. Given this, having a clean, one-pass design is near impossible. If it seems hard to do this right, that's because it *is* hard, not because you're misunderstanding something.

• We often are handling redundant copies of data: there is no single source of truth. For example, data is stored both in FITS files in the filesystem and in the database. The parent-child relationship between bundles and collections, and collections and products is stored as link entries in the database, *and* as containment in a single-versioned filesystem, *and* as entries in `subdir$versions.txt` files in a multiversioned filesystem.

The advantage in this is that you don't need to convert data from another format if you have it in a convenient format; the disadvantage is that the multiple copies of the data can disagree. Keeping them in sync is extra labor. Keep this in mind: if you change one version of the data, you may have to change it in a different place, too.

• PDS4 components seem like tree-structured data, but

once you including versioning, it acts completely different from any other data and your intuition will throw you off-course. Different versions of a bundle, collection or product can have different contents, and a single version of a collection or product can be a child of multiple versions of its parent.

I've found it useful to think of PDS4 component versions and their relationships as completely separate objects. This is what I ended up doing in the database: PDS4 components exist as database objects, but the parent-child relationship is stored as a *separate* table of links.