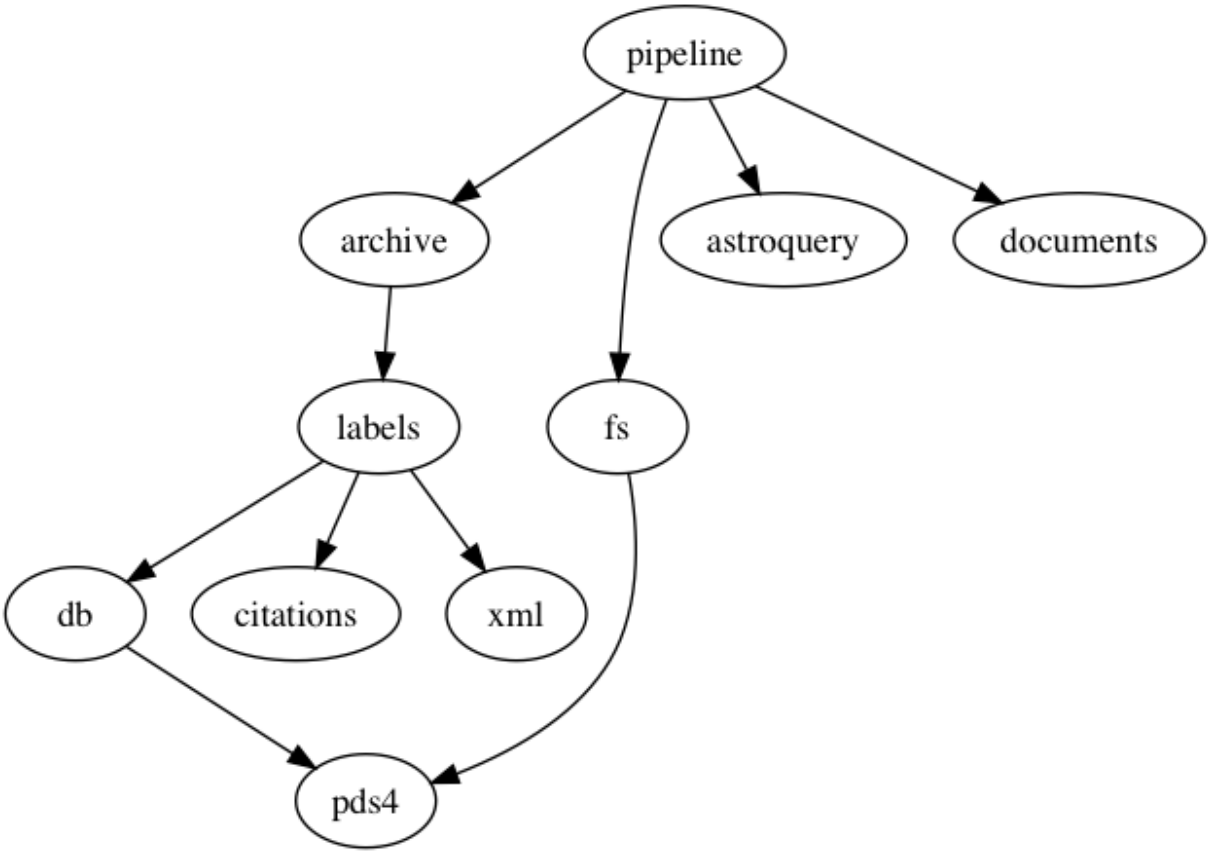
`pdart` assumes a `pds-tools` directory at the same directory level as `pdart`.

You must install `python3`, `black` (a formatter). I used `homebrew` for that. You will also need the PDS4 validation tool, which can be downloaded from the PDS4 site. *URL needed.*

The rest should be automatic using makefiles and the Python tools. Key tools are `black` (formatter), `mypy` (typechecker), and `pytest` (test-runner).

I try to do as much as possible in `Make`, so everything can be automated. This includes formatting and typechecking.

Reduced dependencies:



`pds4` contains basic datastructures and utilities for PDS4 identifiers and HST filenames.

`astroquery` contains functions to download the datafiles from MAST using the `astropy` library. There are a lot of details, so I put them all into one place so I wouldn't have to remember them.

`documents` contains functions to download documentation file from MAST and to convert their encoding to UTF-8 if necessary.

`citations` contains code to parse documentation files

and extract information like author names and publication dates that are needed in the XML label files. Mark wrote this code.

`xml` handles construction, pretty-printing, and validation of XML files. XML files are built from templates, which contain holes that can be filled with single nodes, or lists of nodes ("fragments"). Templates are interpreted by passing in a dictionary that says what should go into each hole.

When the code was first written, we were building the XML as we read in the data, so we needed to build the XML starting at the bottom and working upwards. The standard XML libraries for Python build XML starting from the top, root node, and add children, working towards the bottom. To get around this, instead of building the actual XML, we make and combine *builder functions* starting at the bottom working towards the top, and our final result is a function that, when run, builds the entire XML tree.

There is some ad-hoc code to typecheck what we're building, because it was so easy to make mistakes and so hard to debug.

Later we decided to read all the data into a database first, and then build the XML, so the original way of working with builder functions instead of XML nodes may no longer be the best way to construct the XML.

There are functions to validate an XML file using an XSD file (which validates the structure) and a Schematron file (which validates the contents).

`db` contains the code to interact with the bundle databases. For each bundle, we create a database that contains all the information needed to build the XML label files for the datafiles in that bundle. We use the `SQLAlchemy` library to interface with the database. It lets us define Python classes that correspond to rows in the database, letting us use Python objects in a natural way instead of having to write SQL code.

The classes are defined in `SqlAlchTables.py`. If you want to edit this, you'll need to understand SQLAlchemy. `BundleDB.py` defines an object that represents the bundle database. It has methods for all the basic tasks you must do, and acts as a wrapper around the SQLAlchemy code. You can call methods of `BundleDB` without knowing anything about SQLAlchemy.

`FitsFileDB.py` has functions to put information from FITS datafiles into the bundle database and to get it back out. `BrowseFileDB.py` does the same for browse files (small image files).

Sometimes it's useful to travel over all the parts of a bundle. `BundleWalk.py` gives a framework to walk through a bundle, providing access to the database at all the subparts.

You can implement a subclass of `BundleWalk` by overriding the different `visit_XXX()` methods to do what you need to be done

`labels` contains the code that tells how to build the different kinds of XML label files. It contains templates for the different kinds of files and for parts of the file. They often appear in pairs, with one giving a higher-level view of what's being built, and the other containing the templates used to build. They will have names like `FitsProductLabel.py` (higher-level view) and `FitsProductLabelXml.py` (the templates).

Besides label files, there are also collection inventories, which are an ASCII formatted list of files.

To get information from the database to build label files, sometimes we need to search in a datafile for it, sometimes we need to search in just one part of the datafile, and sometimes we need to search through multiple datafiles. To generalize the idea of searching, the file `Lookup.py` defines an abstract class `Lookup` that gives a single, dictionary-like interface to any of these kinds of searches.

`archive` contains functions to create *PDS4 manifests:* lists of the files to be delivered to our clients. Each file gets a checksum and a physical location within the delivery tarball (since the PDS4 standard does not say where files must live).

`fs` contains *Python virtual filesystems* using the `PyFilesystem2` library. (Documentation here: https://docs.pyfilesystem.org/en/latest/index.html). A Python virtual filesystem is an object with an interface that looks like Python's module `os` and `os.path`. For instance, if you have a filesystem object `fs`, instead of calling `os.open()` to open a file, you would call `fs.open()`. This way, the code is presented with an object that looks and acts the same as the filesystem, but can do more things for you.

In PDS4, a bundle and its files can have multiple versions and we need to keep track of all of them. The `fs.cowfs` module provides a *copy-on-write* filesystem that, from the front, looks like any other filesystem that you can read and write to. On the back, it maintains two directories; one read-only directory contains the original contents, and the other contains only the changes that you've made. This lets us recognize when we need to create a new PDS4 version of the file and lets us reuse the files that haven't changed, but all this complicated logic is hidden from the user of the filesystem.

The `fs.multiversioned` module provides a way to store all the different versions of files and what directory they live in into a single back-end directory structure. The `VersionView` class gives you a read-only view of a single bundle version. You can interact with it as if it's any ordinary directory, and the virtual filesystem will hide all the translation it must do in back to figure out the version

locations and what's in what directory.

The `fs.primitives` module contains an implementation of the virtual filesystem interface that is inefficient, but easy to prove correct. This was used while developing the other virtual filesystems and making sure they were right.

We store files in a logical hierarchy that matches the PDS4 concepts, but for the deliverables, a more traditional, visits-based hierarchy is easier for humans to use. Rather than hard-coding this into the system (since human tastes change), we put the visits-based structure into its own module The `fs.deliverablefs` module provides a virtual filesystem that looks like the logical PDS4 structure in the front, but in the back, it is visits-based. To make a deliverable, you copy the whole logical structure into it, and out the back-end comes a visits-based structure.

The virtual filesystems do a lot of work and let us isolate a lot of complicated logic into a single place. But they are tricky to get right and require good knowledge of the `PyFilesystem2` library to edit. If they are working right, they can be treated as black boxes that work just like actual filesystems.

The `pipeline` module drives all the other layers. If you want a high-level view of what needs to be done to create a PDS4 bundle, this is the place to start. The current implementation is just one way to do it, and I expected

eventually to reimplement it using Apache Airflow (https://airflow.apache.org) to automatically handle things like scheduling, running bundles in parallel, and monitoring from a GUI.

The current implementation runs twelve steps, each implemented in a class:

1. reset the pipeline
2. download documents
3. download datafiles
4. copy the downloaded files into single directory
5. record while files have changed
6. create a directory for a new version of the bundle if any files have changed
7. fill the bundle database with information from the new files
8. build new browse products for changed files
9. build new labels for changed files
10. insert the new bundle version into the archive
11. create a deliverable directory
12. validate the bundle with the PDS4 validation tool