**Floe manual**

revised: 4/5/2018

*What is floe?*

Floe is a Python program that moves and transforms data. It doesn't require human attention. Each time it's run, it makes sure that one set of data is up-to-date with respect to another set of data. Then it stops.

*What motivated floe?*

In 2016, I'd gotten hooked on the Arctic Sea Ice Forums (https://forum.arctic-sea-ice.net) hosted by "neven" (http://neven1.typepad.com), and frequented by many smart people who discuss the current state and future prospects for the Arctic ice cap (and many other topics). User "A-Team", who has posted many unique and illuminating graphics to the forum, began to discuss a new NOAA project (called RASM-ESRL) that publishes daily data sets and forecasts. A-Team invited others to participate, and I made contact. As a programmer, my angle was to see if all the hand work required to make graphics like A-Team was making could be automated. If so, we could create a website that would host versions of these graphics that would always reflect the latest data, without any ongoing human intervention.

I envisioned a program that would execute these steps:

1. Read its configuration and locate an input folder, and an output folder, on whatever computer it's running on.
2. Check for new or changed files (since the last run) on NOAA's FTP server, and download them to the input folder. Recursively unpack any archive files (such as *.tar.gz files).
3. Delete any files in the output folder.
4. Walk the tree of new files in the input folder, and for each file, run a set of rules. A rule has a condition (often based on the name of the file and its location in the tree), and a set of actions to take if the condition is true. (The first action I implemented was simply copying a file from the input folder to a specific location in the output folder. The next was an action to rescale an image before copying it. I've added more actions as needed.)
5. After the walk completes, there will be new files in the output folder. So, walk the tree of files in the output folder, and run the same rules against them. This can in turn create

more files, requiring yet another walk, and so on. Floe keeps track of how many new files are created with each pass; when this number is zero, there's no more work to do.

6. The point of floe is to make these creations available to the public, so the next step is to upload them to a web server. (As a starting point, I chose to use Amazon's S3 service as an affordable and flexible place to host files of all sizes. An S3 "bucket" can be configured as a web server and all the files made public.)

7. The uploaded files need to be organized and presented so that they make sense. This requires some HTML pages with links to the graphical and other products available. Now, S3 as a web server has a particular limitation: it only serves static files. There's no such thing as "server-side code" (pages that are built dynamically on the server when requested). So, after floe has finished building the output tree and uploading it to S3, it then rebuilds all the HTML files making up the site, and uploads them to S3 too. This allows the pages to adapt to the data - there might be five of some particular kind of image today, versus three yesterday - without the more conventional solution of running code on the web server for each request. (This was the initial vision, but not a hard limitation in floe, as we'll see later.)

8. The floe program is scheduled to run periodically (perhaps every few hours), so the website will always reflect the latest available data.

*So floe is just a Python program custom-made for this job?*

No, I quickly realized I wanted floe to be a framework, so its code is generic. Everything that's particular to this task of using NOAA data to make interesting graphics about sea ice is either in configuration files, or in a set of HTML templates that floe turns into web pages or other assets. The same floe program could also be used to do something else entirely, like search for interesting real-estate deals in a MLS database and build out a constantly-updated website of customized listings.

*How does it know about things like scaling image files? Is that all custom Python code?*

No, floe uses external programs to do the heavy lifting. Any tool that runs on the command line, and doesn't require user intervention, can be called from floe. There are many free tools available to do image processing tasks. PanoplyCL [currently in beta, see https://www.giss.nasa.gov/tools/panoply/], ImageMagick, and Gimp are the first three I've connected to floe. There are also lots of useful tools that can be installed as Python modules (SciPy, for example) and those are also easy to hook into floe. Finally, there's a plugin framework that allows new Python functions to be loaded and integrated with the core framework.

## Requirements:

OS: floe can run on linux, Mac OSX, or Windows, but I haven't been running it much on Windows lately, and haven't run it on OSX at all. There are dependencies on standard linux tools such as rsync that would require some attention on Windows (cygwin might be a good solution). OSX, being a flavor of unix, should be more straightforward.

Python: floe is now Python 3 only (version 3.4 or later). It was initially developed on Python 2.7, but that branch is unmaintained.

Boto3: This is the Python library that talks to S3 (and other AWS services) and is the biggest external Python package required. (Not required if not using the S3 destination manager, S3.py.)

Database: floe originally used the file system for persistence of its metadata. This still works, but since data is only written to disk at fixed times, a crash or termination can cause data loss. There is now a plugin under development to support PostgreSQL as the persistence store.

External tools: There is support for PanoplyCL and ImageMagick, and others will be added; however, these are invoked as external command-line processes, so you only need to install the ones you actually intend to use. The floe program will run fine without them. (When you start floe.py, Python will load all the files in the configured plugins directory, and try to satisfy their import statements. If a plugin file isn't needed, and imports libraries that you don't want to install, just move that plugin file out of the plugins directory, and it won't be loaded.)

Virtual environment: Python has a great feature, which is the ability to set up any number of "virtual environments" on a single machine; each can have its own version of Python, Python libraries, et cetera. (This will be discussed in the "quick start" document.)

AWS account: to use S3, you need one. They are free to create, and for the first six months you can use "free tier" services at no cost. Outside the free tier and/or after six months, you pay as you go for pretty much everything, so you need to be aware of that and make sure you don't incur unexpected costs.

Hardware/Hosting: floe should run on any reasonably modern PC, linux web host, or linux cloud instance. Not all hosting is "reasonably modern," though - for example, my HostGator shared web hosting won't run floe, because the Python versions they offer are a bit too old, and since it's a shared machine, you can't install a newer one for your own use.

Network: image files, among other things, can be large. I was trying to develop floe on a typical home broadband setup, and the 10MB/s or so download speed was fine, but the 1MB/s or so upload speed soon became intolerable. So I still do coding at home, but use a low-end Amazon

EC2 instance to do full-scale work, because network bandwidth to and from that instance is much better.

A separate “quick start” document will walk through a complete install and test run of floe, so further details will be there.

## Core concepts:

*Input folder*: the configured local folder where external files are downloaded for processing by floe.

*Output folder*: the configured local folder where floe places the files it builds so that they can be uploaded and merged into an existing destination.

*Rules*: as stated above, a rule has a condition (often based on the name of the file and its location in the tree), and a set of actions to take if the condition is true.

*Worklists*: a worklist is a named, ordered collection of work items. A work item is a reference to one or more files, to which various kinds of metadata can be attached.

*Expensive* vs. *cheap* files: "expensive" files (a.k.a. "content") are files downloaded from external sources or derived from them, and are always assumed to be potentially large and/or costly to re-create, so floe does its best to never download, build or upload these files redundantly. "cheap" files are assumed to be small files, usually containing text, which are assumed to be cheap to re-create, so floe doesn't care if it creates the same file a number of times during a run.

## Sequence of events in detail:

Floe executes a series of main operations in a fixed order. (You can use the config file to skip some of these steps if necessary.) The implementation is pretty close to the initial vision outlined above:

1. Acquiring new or changed files;
2. Applying rules to the files, and to new files created by rules, until there's no more work to do (and populating worklists along the way);
3. Upload the files we want to a destination manager (such as a web server or S3 bucket);
4. Using a set of templates to render a set of supporting files (such as HTML files comprising a website);
5. Uploading the supporting files.

Now let's look at those steps in more detail.

**1. Acquiring new or changed files** from external sources (such as FTP servers). Changed files are detected using whatever metadata the source provides (for FTP, the file modification timestamp is used). These are downloaded and placed in the input folder (in subfolders per source, to keep track of the origin and prevent possible name collisions). This is handled by the **Input Manager** (or IM) component. Note that floe is fundamentally file-based, and no matter what kind of source the data comes from, the end result must be a plain file-system file in the input folder. Floe is not meant to handle live streams.

Floe tracks file metadata for downloaded files (which is persisted between runs), and uses this on the next run to determine what, if any, files need to be downloaded. This means that old, already-processed files don't need to be physically kept in the input folder, which decreases pressure on disk space.

A component called the **Tree Processor** (TP) now takes over and executes these steps:

**2a. Iterate (or "walk") over all the files and folders** in the input folder, applying a set of rules to each file. Rules consist of a **match** condition, which determines whether the rule applies to a particular file (based on its name and/or other attributes), and one or more **actions**, which are processing steps using the current file as input. It's important to know that this pass over the input folder only happens once per floe invocation (or iteration, described below). That means that rules in this pass generally do something that results in a new file being created in the output folder. This could be as simple as copying a file to a certain location in the output folder, or the file might be passed to a program that processes the input and creates a different kind of file in the output folder. (For example, the PanoplyCL program takes a netCDF data file as input, and produces an image such as a .png file as output.)

If archive files (such as *.tar.gz) are encountered during this walk, they are unpacked. This is recursive, and the files go into an __UNPACKED__ folder inside the input folder, where the TP immediately iterates over them before getting back to the main walk. (Any folder structure within the archive is preserved within the __UNPACKED__ folder, to avoid name collisions and so that you can reference these paths in rules if desired.)

**2b.** The TP now turns its attention to the output folder, which was emptied at the start of the run, but now has files in it as a result of step 2a. **The TP walks over the output folder**, and again applies the same set of rules as it encounters each file, and if the match condition is satisfied, takes the specified actions. (Although it is the same set of rules, in practice, the rules that match here are typically not the same ones that were applied in step 2a - the rule syntax makes it easy to check where the current file is sitting, if desired.)

Any new files created during this step must themselves be placed somewhere in the output folder.

**2c.** Now that there are new files, we might like to be able to apply rules to those files. One way to do this would be to track each new file created. The inconvenience that would create is that when we invoke an external program, we don't automatically know whether it created any files, where they were placed, and what their names are. So the plugin code that invokes any external program would have to keep track of what files were created and report that back to the TP. This would make writing new plugins more difficult.

What the TP does instead is to simply **walk the tree again, looking for new files**. They way that it avoids processing the same file more than once is to add a 'rules_run' tag to the metadata for that file. When the file-system walk finds a file without corresponding metadata, it knows this is a new file created by a prior process, and runs the rules against that file.

This is repeated until TP has made a walk (or "pass") over the files without finding any it hadn't already processed. (To avoid an infinite loop, there is a configurable limit [default 10] of passes after which an exception is raised, halting the run. During rule development, it's not hard to accidentally write rules that keep making new files forever.)

**2d.** At this point, floe looks for a special file named **'run_post_tree'**, and if it exists, some additional work happens. This option is provided to take care of situations described in "Multiple Dependencies" below. Detailed discussion is deferred to that section [TODO WRITE THEM!], but briefly, there might be actions that depend on more than one input file, so they can't be safely run until we know the TP has finished its work and all those input files are built.

**2e.** The rules that TP has run so far are of the default type (called 'self-tree' because they operate within the output folder tree). Floe also supports another type of rule, called a 'dest' or **destination rule. Now it's time to run those rules** (if there are any - they're optional).

They work a little differently than the ones we've covered. They don't look at the output files - instead, they look at the tree of metadata representing the destination, as it *will be* after all the files have been uploaded (step 3). This is possible because, whenever the TP finds a new file and runs rules against it, one of the things it does is notify the destination manager, which notes the file with a status of 'pending'.

The purpose of all this is so that actions can be taken based on the full final state of the destination (e.g. to facilitate building web pages that match that final state).

'dest' rules are not allowed to create any new files. Usually, their action is to add something to a worklist, for use later (in step 4).

Rule actions, regardless of the rule type, can also attach arbitrary metadata (as key/value pairs) to any file. For example, when executing the "make_thumb" action to make a thumbnail of an image, floe attaches a metadata item 'thumb_full' to the parent image, pointing to the thumbnail

file. It also attaches 'parent_full' to the thumbnail file, pointing to the image file. This allows links between the two to be rendered, without any special naming convention for thumbnails.

When files are uploaded to the destination manager, this per-file metadata is handled appropriately. For instance, destination manager content is organized by key, rather than file path, so the thumbnail information just described is rewritten appropriately as 'parent_key' and 'thumb_key'. Metadata attached by the user (by rule actions) will generally pass through to the destination manager unchanged.

Now the **Destination Manager** (DM) takes over to:

**3. Upload the files** in the output directory to the destination endpoint. This is a merge - we add any new items, replace any existing items with new versions, and leave everything else alone. The directory structure in the output directory is mirrored exactly in the destination tree, with no changes. (Rules are flexible enough to make the output directory structure look exactly as you want it to.)

This upload process may be slow (as for large image files sent across the internet to an S3 bucket, for instance). So the DM takes pains to avoid uploading any file that's already in its destination store in identical form. To do this, its persistent metadata store saves (among other things) the md5 hash of each item. Before uploading a file, it checks that against the md5 hash of the new file with the same path and name, which was computed earlier, when the new file was created. If the md5 values match, the file is skipped.

Another point about the DM upload: during the TP step, intermediate files may be created that shouldn't be uploaded. For instance, an animated .gif might be assembled from individual frames; only the finished .gif should be uploaded. To support this, anything in the output folder whose path starts with "tmp/" will be ignored by the DM. So the user, via rules, can use that folder for intermediate files. Floe's default behavior is to empty the output folder at the start of each run, so the user doesn't have to worry about cleaning up these files.

Here, the DM hands off to the **Webmaker** (WM) component.

(Note: the floe code still contains names like "webmaker", which imply that floe is limited to building web pages/sites. Some of these are being weeded out over time; others might be kept for simplicity.) Up until now, everything we've done has been directly driven by the new/changed files we downloaded from outside sources. In particular, if there were no new/changed files, we will have done nothing. All the files we've processed are in the "expensive" category; we assumed that every file might be large, and every processing step slow, so we've taken care to do only the work that was necessary.

Now we shift gears. We will build a set of files we consider "cheap" -- assumed to be small and quick to process. And we will build these files (some of them, at least) regardless of whether we have new data or not.

This is confusing in abstract terms, so here's the motivating example: suppose you are downloading large data files, updated daily, and building large images from them by some computationally expensive process. Now, you want to host those images in a web page, showing perhaps the most recent 10 daily images.

The process up til now has ensured that we build new images as needed, and stay caught up with whatever source we get the data files from. The older images are still safely stored in the destination where we uploaded them (in step 4, see below) because we don't want to rebuild them if they're still valid.

But what about the HTML page to show the latest 10 images? We need to rebuild that any time we've created a new image. And that page needs to know not only about the new image, but also about the ones already in the destination from past runs. So it has to have access to complete metadata about what's in the destination.

And what if there aren't any images to show at all? (Maybe there's been a hiatus at the source, and they have no data 10 days old or less at the moment.) Do we just not build that HTML file, so that on our website it's either missing (a dead link), or shows obsolete data? Rules in the TP are only triggered by the presence of new/changed files, not by their absence.

My solution to these issues is based on the fact that building the actual HTML is cheap, and the files are small. So, rather than having to write more logic to control precisely when to rebuild each HTML page or section thereof, it's easier to just rebuild all the HTML files for the site, so everything is in sync.

**4. Render files from templates**: this starts from the files in the admin/templates/site folder. This folder can have subfolders, and for every file in this tree, a corresponding file will be built in the output.

Here's the simplest possible example: suppose you have a template named index.html in the site folder. And suppose this whole file is hard-coded HTML, with one exception - in the page text, it says "Site last updated [update_time] UTC." The square brackets are floe's template syntax. Floe will copy the template index.html to a real index.html in the output folder, but along the way, it will evaluate all those bracket expressions and replace them with something else. In this case, the floe code knows that [update_time] should be replaced with, for instance, "Mar 06 2018 23:03:13" in the copy it writes.

An even simpler example would be static files (perhaps .css or .js) that don't need to be changed at all. Seeing no bracket syntax in them, floe will just copy them to the output as-is.

Floe can interpret a number of different kinds of bracket expressions. A more realistic index.html template might look like:

```
[part name: head, title: Interesting Climate Images]
[part name: top-menu]
[part name: welcome]
[part name: foot]
```

"part" tells WM that it should look for a template in /admin/templates/parts with a matching name (head.html for the first line) and substitute that in place of the bracket expression. In this case, there's also a parameter 'title' with a value to be used. This is recursive - the head.html part template will probably have its own bracket expressions, which will be interpreted.

This way of building web pages out of reusable parts is used in all sorts of web-building frameworks. Most of those frameworks are dynamic - the templates aren't filled in until a page is requested; but floe, and some others, are static - all the HTML is generated during the floe run, and when the page is requested it's simply served as-is.

Another kind of bracket expression that floe can handle is worklist invocations. Recall that populating worklists is one of the things that rules can do. Here is where we can use them to customize our HTML. A worklist invocation looks like this:

```
[worklist name: popular_images, mode: thumbs, <...other arguments>]
```

The word "worklist" at the start tells floe this is a worklist invocation. This might be used to render the most popular images on the home page, as thumbnail images which can be clicked to show them full-sized. Floe will look for a worklist named 'popular_images', and will render the HTML snippet to show them as thumbnails (the "mode: thumbs" argument routes the request to the correct floe plugin function for that).

Worklist invocations like this offer the most flexibility in inserting whatever you want into a template, and they form the connection between the "expensive" files built from new/changed data in earlier steps, and the "cheap" files that WM builds (which don't have to be HTML, though that's the most obvious application).

Inserting text into an HTML page isn't the only thing a worklist invocation can do. It can also generate a set of new HTML pages from the items on a worklist. For example, you might have a set of images and want to have each one hosted in its own HTML page, where additional information can be shown. The worklist invocation for this would specify "mode: pages", and specify a page_template parameter. Floe would look in /admin/templates/generated for a corresponding template, and create one new HTML file from that template for each work item on the worklist.

You can also think of a worklist invocation as a generic function call into floe code. If the 'mode' argument matches a floe function - most likely a plugin - then you can call that function, with some arguments inside the square brackets, as well as a list of one or more work items created earlier.

I've left out another advanced feature, in the interest of focusing on the main logic first. The very first thing the WM does is actually to check for another special, optional file, named 'run_pre_web'. If it exists, it is rendered before any other template. The rendering process is the same as any other template, but no output file is created - it's processed only for its side effects. If a 'run_pre_web' file is used, it will probably contain only a series of worklist invocations. This is useful for website scenarios that require doing several things for a set of images or other content files - you might want to generate a separate HTML page for each of those files; create a list of links to those files which can be put in an index page; and choose one of those links to be the default. In this case, you can use 'run_pre_web' to call a floe function which will go through a master worklist, and populate other worklists for these separate tasks. This sort of advanced functionality is easier to show in a tutorial example than it is to explain (the customization code for floe.keytwist.net will be posted soon as an example).

5. Uploading the supporting files (the "cheap" ones created by WM): WM maintains a list of all the files it's created during template processing. Since they're "cheap", they are just unconditionally uploaded via the DM.

Finally, at the end of a run, the output files are copied to a local archive directory (specified in the config file). (The magnificent unix 'rsync' utility does the work.) If everything has worked properly, then at any point in time, the archive folder will be a faithful copy of the items in the destination store. This allows historical files (which weren't build in the current run, so they're otherwise not available locally) to be used as input for processing. Without the archive, these files would have to be downloaded back from the destination store, which might be very slow. The archive can also be used for disaster recovery if for some reason the destination store is deleted or corrupted.

## Configuration:

Like many service-style programs, floe gets most of its settings from a text-based config file. The first command-line argument for floe is the config file, e.g.:

```
python floe.py configs/dbg/dbg.config
```

If no config file is specified, the program just prints an error and quits, because the possible consequences of running the wrong configuration outweigh the convenience of having a default.

You can include other files into your config file by using << notation, for instance:

```
<<include ../local/paths.config
```

The lines in the referenced file are added to the referencing file before any parsing is done. This feature is not recursive at present; nested includes will cause a syntax error. Note that this same notation can be used in a rule file.

The config file layout is sections in square brackets, with each section containing key = value entries. Spaces around the equals sign are ignored, and recommended for readability. Blank lines or lines where the first non-space character is '#' are ignored. Many entries are optional, with defaults defined in the program. Let's go section by section (they don't have to be in order in the file):

### Actions section:

```
[actions]
ftp = False
process = False
upload = True
web = True
archive = True
```

The `[actions]` section is an overall control panel, a quick way to turn steps of the process on or off for a given run. In production, everything would be set to True (or left out altogether - all these default to True). Note that "True" and "False" (with capitals) are what Python uses, but in the config file we'll also accept "true" and "false" if you prefer, or forget.

`ftp` is the step that fetches a contents listing from the ftp server(s), and if files are new or changed, downloads them a local folder (the "input" folder, see below). This folder is not emptied between runs, so for testing, you can turn this step off and no files will be downloaded.

`process` is the step that walks the input files, applies rules, and continues the process in the output folder until no more changes are made.

`upload` is the step that uploads files from the output folder to the DM.

`web` is the step that turns all site templates (explained shortly) into web pages (html, stylesheets, javascript files, etc.) and uploads them to the DM. It is also the step that processes "worklists" (also explained shortly), which can cause new files to be built in the output tree. Finally, it uploads all these files to the DM.

`archive` copies all the files in the output folder to a separate archive folder (on a local drive). Old archive contents are not deleted if they aren't in the output folder, so the archive folder

should normally be a perfect mirror of what's been sent to the DM. The archive has several uses, but not all configurations need it, so it can be turned off.

### Input section:

```
[input]
servers =
    ESRL_FTP_SITE, ftp1.esrl.noaa.gov, anonymous, me@somewhere.com
    MY_SITE, ftp.myplace.com, me, my!Passwd.

folders =
    MODEL_OUTPUT, ESRL_FTP_SITE, RASM-ESRL/ModelOutput
    MY_FOLDER, MY_SITE, public_ftp/my/sandbox

urls =
    DEMO,
    https://x.com/image[YYYY][MM][DD].jpg,
    "",
    2018-03-01, 0, fetch
```

The `[input]` section governs the downloading of files from FTP or HTTP servers (or other sources, in the future). First we have the `servers` entry. Note that this, and the `folders` entry following, are multi-line. (The indentation is for readability and is not required.) This is how a list of multiple values can be entered for one key in the config file. We need that here, because we can configure multiple FTP servers to be read from, and one or more folders within each of those FTP servers.

The `servers` entry defines what FTP servers we'll check. In this example, we've defined two. One is the actual NOAA ESRL site where we get Arctic data, and the other is a test server for development and testing. The comma-separated fields on each line are:

1. A tag for the server, which can be anything you like (no spaces; best to stick to letters, numbers, and underscores). I use all capitals, but that's just for my own mental housekeeping.
2. The actual URL of the FTP server.
3. The username to log in with. The ESRL site allows anonymous logins, so by convention, the login is 'anonymous' and the password is the email of the person logging in. If there is a real login and password, the password is here in plain text, so be careful (or fork the floe code to handle it differently).

Next comes the `folders` entry. FTP sites often have a lot more information than you'd ever care to download, so floe is set up to only download specific folders on any of the configured sites. (Note: this is NOT recursive; subfolders on the FTP site are not downloaded, by design. A flag to turn on recursive downloading would be easy to add.)

So, each configured server can have one or more folders to be read and downloaded. In this example, each of our servers has one configured folder. (Side note: you can comment lines out in the config file by putting '#' in front of them, which often comes in handy.) The fields in a folder entry are:

1. A tag for the folder, which is anything you want. Note that this will be used as the name of a destination folder within the input folder where files from this FTP folder will be stored. We could just dump them all into the same input folder, but that would lead to (a) possible name collisions, and (b) possible confusion about which FTP folder/server a particular file came from. So we segregate them.
2. The tag for the FTP server where this folder is found - this refers back to one of the tags you defined in the `servers` entry.
3. The path from the root of the FTP server to the folder of interest.

The next entry, `urls`, governs the downloading of files by URL. Since HTTP{S} endpoints don't have an inherent, readable tree structure like FTP servers do, the configuration for URLs isn't built around servers and folders; it's just a flat list of url entries, all independent. Note that the fields can be on separate lines, as shown here; the lines are merged on comma and treated as if they were a single long line.

The fields in a url entry are:

1. A tag for the entry, which will be used as the folder (a child of the input folder) for all files downloaded using this entry.
2. The URL template. This is a URL with placeholders in brackets. The placeholders determine how to generate a series of URLs from the template. The only mode currently implemented (and the most likely to be used) is date mode, where there is a distinct URL for each calendar date. In the example above, the template is `https://x.com/image[YYYY][MM][DD].jpg`; the year, month and day placeholders are filled in with successive dates, ranging from the start date (see below) up to the present date plus one (to allow for time zone differences and pre-posting). An attempt is made to download each generated URL. Another mode that will be added later is date/time mode; this will support multiple downloads per day. Yet another mode would involve scraping a page for links and attempting to download those links. As usual, only files not already on the server will be downloaded.
3. Destination pattern: this is optional (use "" if not needed). If present, this is an expression for the subpath (under [input_root]/[tag]) and name under which the file will be stored. This is needed in cases where many files will all have the same name (e.g. "Arctic10.gif") because the encoding of the date is in the path, not the file. To avoid overwriting the file over and over, this pattern can be used, and supports the same placeholders (e.g. [YYYY], etc.) as the URL template. The destination pattern can also be used to simplify complex file names.
4. Start date: in date mode, this is the earliest date (in YYYY-MM-DD format) that will be used to generate a URL from the template.

5. Delay: the delay (in seconds, a nonnegative integer) between requests to the same URL template. This allows throttling to reduce server burden or to prevent being labeled as an attacker. The current implementation is a simple time.sleep() call; if a future scenario justifies it, a better solution would be a threaded implementation that can download in parallel from many URL templates while honoring the delay for each.
6. Download method: this can be:
    a. `fetch`: a simple GET request that, on success, will save the returned data into a file. This method is implemented.
    b. `snap`: request a snapshot (screen grab) of the page as it would render in a browser, using Google's Puppeteer (a headless Chrome browser). The plugin for Puppeteer exists, but a little more integration work is needed to support this mode.
    c. `text`: not implemented yet, this will GET the page and then use rules to extract key/value pairs of interest from the text.

Here are some more `[input]` entries, all optional (moved down here so you don't have to jump back and forth so much):

```
download_limit = 1
url_download_limit = 1
include_files = .
exclude_files = 4NIC
metadata_from_local = True
```

`download_limit` is a way to limit the number of files downloaded per configured folder. The default is 0, which means no limit. `url_download_limit` is similar, but applies to the URL IM

`include_files` is a regular expression; file names that match the expression will be included in the download. By default, all files are included. This only applies to FTP downloads.

`exclude_files` is also a regular expression (floe uses them a lot), but files matching this one will, as the name suggests, be excluded from the download. Again, by default everything is included. You can use both `include_files` and `exclude_files`; in that case, each file name has two chances to get kicked out from the download. Only applies to FTP downloads.

`metadata_from_local` is used for testing/debugging. It will be discussed later.

### Local section:

```
[local]
input = /floe_data/configs/dbg/input
output = /floe_data/configs/dbg/output
admin = /home/floeuser/floe/configs/dbg/admin
archive =/floe_data/configs/dbg/archive
```

```
plugins = /home/floeuser/floe/dev/plugins
user = /floe_data/configs/dbg/user

logfile = dbg.log
loglevel = debug
log_to_console = true
local_store_start = 2017-06-01
debug_tags =
       ALL
       COPY_WITH_METADATA
```

The local section defines the important places on the local drive(s) where floe keeps things. You can have multiple floe configurations doing different things (on the same, or different data sets) on one machine, so it's smart to have a naming convention to keep things straight. In the four path entries above, I've shown two naming conventions: (a) I keep multiple configurations under a "configs" folder, and (b) I'm keeping the relatively small "admin" folder under my home directory, and the other three (which might become quite large) on a separate drive named "floe_data". But you can lay things out any way you need to.

`input` is the root of the input folder, where as discussed, input files are downloaded from FTP, URL, or other sources. The input folder is not automatically cleared, so it acts as a cache, in case you need to reprocess some previously-downloaded files, but want to avoid downloading them from the server again. (This is one of many "state management" challenges that floe tries to address, more discussion later.)

`output` is the root of the output folder, where the system builds exactly, and only, the set of files that are to be uploaded to the DM. There is one exception to this statement - if the output folder contains a folder named "tmp", files in that folder will never be uploaded. This allows the creation of temporary files, against which further rules can be run, which will automatically be deleted at the end of the run.

`archive` is the folder to which the files in the output folder (excluding tmp/) are archived at the end of each floe run.

`admin` is the "brains" of a particular configuration. By convention (only), it contains the config file. It also contains template files, persisted state files, and log files - pretty much anything housekeeping-related. There is no reason for the admin folder to ever get very large (although the log files do bear watching).

`plugins` will be scanned (recursively) when floe starts up. Any file with a .py extension will be imported as a Python module, and a function named 'register' in that file will be called. That method should return a dictionary mapping function names to function objects. This will allow floe code to call any of those methods by name. One example of how this is used is when parsing actions; if a rule says

```
if <some condition>:
  foobar dest: [$1].png, color: red, size: 300
```

then floe will look in its plugin registry for a plugin function named 'foobar' and will call it with the parameters listed in the action; the string 'foobar' doesn't appear anywhere in the core floe code, nor does floe know what sort of parameters the plugin function requires. This makes it easy to extend floe without changes to the core code.

`user` is the location where per-user server files are stored (this is beyond the scope of this document).

And the remaining entries in the [local] section:

`logfile` is the name of the logfile to create on each run. (It's always located in the admin/logs directory.) The naming convention I use is <config name>.log. A new log is created at the start of each run, but rather than overwriting the old one, it's renamed to <name>.log.<n> where <n> is an integer. The lowest integer that doesn't conflict with an existing file is used each time. So, <name>.log.0 is the oldest backed-up log, <name>.log.1 is the next oldest, and so on. If the number of old log files reaches 500, an exception is thrown. If `log_to_console` is true, log output will be echoed to stdout.

`debug_tags:` Because it can be tedious to debug on a server, floe has an elaborate set of logging flags you can turn on - to see more information - or off - to hide unwanted noise. These are in the `debug_tags` section. Just comment out (with #) any you want to turn off. The tag names are printed in the log, so you can use grep, less, etc. to filter on them. If the special tag ALL is in `debug_tags`, all messages will be printed regardless of what else appears in this entry.

Also, tag names in the config file are treated as prefixes. For example, `RULE_MATCH_myrule` will only report when a rule with that label matches a file, whereas `RULE_MATCH` will report when any rule matches a file.

`local_store_start` is a date in the format YYYY-MM-DD. It should be set to a date before the oldest data this configuration will see. The exact value isn't critical; this date is used when floe is searching backwards through its data collections to find, for example, the 40 most recent instances of a particular type of file. There may not be that many, so `local_store_start` acts as a backstop to make sure the search terminates when there's no more data to be found.

### Process section:

```
[process]
rule_file = demo_rules.config
```

```
clear_first = true
template_root = admin/templates
always_unpack = true
root_file_limit = 2
unpack_files_wanted = Arctic10
template_extensions =
       html
       js
       asp
       tex
```

This section controls the tree processor (TP), which applies rules to files. It's important to understand that in the first pass (pass 0), the TP walks the files in the input folder; in all subsequent passes, the TP walks the files in the output folder (until no work occurred during a pass).

`rule_file` is the name (and path if needed) of the text file containing the rules (see Rules section).

`clear_first` is optional and defaults to True, which means that the output folder is emptied at the start of tree processing. Since the intention is for the output folder to only contain new or modified files at the end of the run, True is what you want, aside from troubleshooting/debugging purposes.

`template_root` is optional, and defaults to admin/templates, which is a sensible place for templates to live. Under `template_root`, `site` contains templates which will always be instantiated (the special files `run_post_tree` and `run_pre_web`, if needed, should go here). `parts` and `generated` are standard subdirectories (but optional if not needed). Other subdirectories, like `panoply` and `puppeteer`, are plugin-specific, and future plugins can designate their own template storage in the same way - the core floe code will ignore them.

`always_unpack` is optional and defaults to True. True means that any time the tree processor encounters a .tar.gz file, it will automatically unpack it into a directory called __UNPACKED__, under the input folder. It will then interrupt the current file walk and immediately walk the files just unpacked, and will do the same recursively if there are any .tar.gz files in that group.

Because unpacked archives can consume a lot of space, and because unpacking is relatively fast, the files in __UNPACKED__ are deleted at the end of every run.

A common situation is that an FTP site, or URL template, has some compressed archives such as .tar.gz files that you want to download, unpack, and process. If so, having `always_unpack` set to True will work well. It would be unusual for .tar.gz files to exist in the output directory at all, but in some use case you might have a reason to custom-build them there, so you can host

them on the web site for visitors to download. That's a case where `always_unpack` might not be a good thing. If you turn off `always_unpack`, you can write explicit unpacking rules to customize when, and to where, unpacking happens.

`root_file_limit` is another way to restrict the amount of work floe does per invocation. "Root file" in this context means a file that is downloaded directly from an input server (as opposed to files unpacked from an archive). If your IM source contains large archives that lead to a large volume of work when they're unpacked, you might use this setting to only download and process one, or a few, such archives per floe run.

`unpack_files_wanted` is a way to customize the built-in unpacking behavior. If this option (which is a regex) is provided, then only files whose names (including path within the archive) match that regex will be unpacked. This can save a large amount of time and space if you only need a few files each from a large archives.

`template_extensions` is list of file extensions that should be considered "cheap" (see above) and which will therefore can contain bracket expressions to be processed. If not specified, the default set (obviously aimed at website-building) is: html, css, js, txt, json, and no extension at all. Tests are case-insensitive (foo.txt and foo.TXT are both treated as template files, if 'jpg' is in the template_extensions set).

### Tools section:

```
[tools]
panoply_workdir = /home/floeuser/Panoply/PanoplyCL
```

"Tools" in this context are Python plugin methods that are invoked as rule actions - sometimes they are wrappers for external applications, sometimes not. The [tools] section is reserved for settings needed by these tools. In the example above, we specify the working directory to use for the "panoply" tool, which invokes PanoplyCL. A naming convention that uses the tool's name as a prefix is sensible. The core floe code ignores this section.

### Deployment sections:

```
[build]
# for S3...
bucket_name = climate-analyst-dbg
region = us-west-1
# for file system dest...
file_dest_root = /floe_data/build
skip_refresh_if_tree_unchanged = true
# in either case:
max_upload_size = 0
refresh_dest_meta = true
```

```
ignore_missing_persist_file = true
```

These sections (`[build]`, `[staging]`, and `[production]`) control the upload manager behavior. They all have the same available options. Only `[production]` is required, and if you want floe to upload files to a final destination right away, just fill in that section. If you want a more production-worthy setup, where files are built to a build location, copied to a staging location, tested, and deployed to production if tests pass, then use all three sections. (See "Deployment" below for more details.)

These settings apply regardless of DM type:

`max_upload_size` limits the size (an integer number of megabytes) of files to upload; files larger than that size are ignored during the upload step. 0 is an allowed setting and will prevent all uploads, which could be handy for testing.

`refresh_dest_meta`, if true, causes the DM to verify the contents of the destination content store and correct any discrepancies with the persisted metadata about those content items. A true setting can cause a long delay, especially for the file_dest DM, because instead of assuming that the destination path hasn't been modified by some other process, it verifies the MD5 hash of each file. (The S3 DM really ought to do this too, but there's no way to verify the MD5 on the server, so this would entail downloading each S3 file in order to calculate its MD5 hash - even more painful.) If false, the metadata is trusted to match the actual files - this is much faster, but you should be certain the files haven't been touched by anyone or anything other than this floe configuration, as mismatch would cause hard-to-debug errors.

`ignore_missing_persist_file:` in the current floe code, metadata is persisted in a flat file containing one JSON blob per content file. If this file can't be found, normally a fatal error occurs, but by setting this value to true, floe will rebuild the metadata file from what it finds in its store of content files. This is handy for experimenting, but be aware that some metadata (such as the link between thumbnail images and their parents) can't be reconstructed, which is why the error is fatal by default.

There are currently two kinds of destination manager (DM): S3, and file_dest. Floe determines which to use by what settings are present in these sections.

For the S3 DM, the bucket name and region are needed by AWS to identify where files should be uploaded. Boto will also need your AWS credentials (one way to specify these is by setting the AWS_SHARED_CREDENTIALS environment variable; see AWS docs for details).

For the file_dest DM, file_dest_root is a location in your file system where files should be "uploaded" (copied) after floe builds them. It's assumed that floe has write permission to that location. When using this DM, this setting applies:

`skip_refresh_if_tree_unchanged:` the tradeoff of `refresh_dest_meta` is discussed above; this setting offers a compromise. The modification time of each file in the destination tree is examined to determine the last-modified time for the whole tree. This value is stored in the metadata file each time it's saved. On reading the metadata, if the last-modified time matches the persisted value, then it's highly unlikely that the tree has been tampered with since the last save, and the metadata can be trusted. (If the values don't match, a full but slow metadata refresh is performed.) This setting can save lots of time, relative to computing the MD5 of every file, at only a very slight loss of safety.

## Regular Expressions:

Regular expressions (a.k.a. regexes) are widely used in floe. They've already been seen in the config file description, and are also used in rules and templates. Floe uses Python's regular expression syntax, which is virtually the same as in Perl and many other languages. Regular expressions can be daunting at first; fortunately there are lots of tutorials on the web.

floe adds some "regex helper" expressions to make reading and writing regexes a little simpler. For example, a regex to match a date in YYYY-MM-DD format could be written as `\d{4}-\d{2}-\d{2}` or in other ways, but in floe you have the option of just writing `[date]`. (NOTE: not yet implemented!) Which brings us to:

## Bracket Substitution:

Both in rule files, and in templates, "bracket substitution" is an important means for customizing behavior. Bracket substitution simply means finding square brackets like [this] within a string, and replacing them with something else, based on interpreting what's inside the brackets. As a simple example, suppose we have a simple dictionary of key/value pairs that maps "programmer" to "Andrew", and "customer" to "Gina". If we have (perhaps in a web template) this text:

```
A meeting between [programmer] and [customer] should be arranged.
```

Then bracket substitution would result in this finished form:

```
A meeting between Andrew and Gina should be arranged.
```

Note that the brackets were "consumed", i.e. don't appear in the new string. This general approach has been used in many tools over the years, especially for web-page templatizing. The delimiters vary - sometimes they're <>, or {}, or {{}}, or <% %> - but the idea is more or less

the same. One thing that floe does differently from some of these systems is that if it can't interpret a bracket expression, it leaves it alone. So you can use literal square brackets in a string like this:

```
Invented by Jones [see http://programming.com/2017/10/01/how-to.jsp], this
technique...
```

and if floe can't make sense of the bracket contents, it will leave them alone. Interpreting what's in the brackets depends on context, and is often much more than a simple symbol lookup.

## Rules:

Rules, along with templates, are the core ways used to configure floe to do some particular job. Let's look at an example rules file. Like the config file, it's plain text, ignores blank lines and lines where the first non-whitespace is '#', and indentation is optional. A rules file contains 1 or more rules, and when the TP is walking files, the rules are applied to each file in the order listed in the rules file.

A rule consists of a boolean condition to test on the current file, and a list of one or more actions to be applied (in order) if the test returns True. For example:

```
if [full] like meteogram\.png:
    stop
```

The first line is the condition, which will always start with the word "if", and end with a colon (:). This condition contains "like", meaning it's a regular-expression condition (there are other kinds too). These conditions have the form: "if A like B:", where A is some string that relates to the current file, and B is a regex to be tested for a match against that string.

In the example above, `[full]` is a bracket expression that means the full path + name of the current file. If that contains "meteogram.png", the condition evaluates to True. (Note that "." in a regex means "any character", so to specify a literal period we have to put a backslash in front of it - or use the floe helper `[dot]` if you find that more readable [TODO: NYI].

The line(s) after the condition are the actions. In this case, there is one simple action, which is `stop`. It prevents any more rules from running against the current file. Since this is the first rule in the file, it guarantees that no file that has "meteogram.png" in its name (or path) will be processed, regardless of what other rules are defined.

Here's another rule:

```
if [full] like [input_root]/MODEL_OUTPUT/REB_plots[dot]([date])/([date)/([any])[end]:
    # copy to history
```

```
    copy to [output_root]/daily/[$1]/[$3]
    # extract frame 0 from gifs
    extract_frames out_dir: [output_root]/daily/[$1]/frame0, frames_wanted: 0
```

There's a lot more going on here. First, the condition is now using a bracket substitution in the regex itself - `[input_root]` is a symbol that gets replaced with the actual input root, which in our example config file above was `/floe_data/configs/dbg/input`. So we are being very particular about the file path - it has to match all the way from the input folder, not just somewhere in between. MODEL_OUTPUT is the folder label for the ESRL FTP folder we also defined in the config file, so that will be in the path too. We want a subfolder with a date, such as "REB_plots.2017-10-01", followed by another date folder, and finally a filename like "Arctic10.gif" at the very end of the filename. "[any]" is a regex helper for ".*" and means "zero or more of any characters" - a true wild card.

We also have some parentheses in the regex, such as around the "[any]" part. Why? Those are match groups, and the parentheses mean that we'll be able to refer later to what was in them in this particular filename. The pairs of parentheses are given symbols in left-to-right order: $1, $2, $3, etc. (as many as we want). See any regular expression reference for more explanation. In floe, we can use those expressions in actions by putting them in square brackets, as [$1], [$2], and so on. You see this used in the actions for this rule.

What are the actions? There are two in this case (along with some comment lines). The first one is a simple file copy. The file that we matched gets copied to "`[output_root]/daily/[$1]/[$3]`". [output_root] is a bracket expression for the output folder, [$1] will be replaced by the first date that we matched (in the first set of parentheses) in the filename, and [$3] is the filename proper, such as "Arctic10.gif". So if the full input file was this:

```
/floe_data/configs/dbg/input/MODEL_OUTPUT/REB_plots.2017-10-01/2017-10-01/Arctic10.gif
```

then that file will be copied to this location:

```
/floe_data/configs/dbg/output/daily/2017-10-01/Arctic10.gif
```

When uploaded to our web site, the relative part of this path (namely daily/2017-10-01/Arctic10.gif) will be the location of the file.

Now the second action will run. This is the expression:

```
extract_frames out_dir: [output_root]/daily/[$1]/frame0, frames_wanted: 0
```

This is how you call a plugin function, in this case the one called extract_frames. You can think of it as a function call, in normal programming terminology. We specify extract_frames as the action, and then pass it a number of key: value pairs (separated by commas) to tell us what specific thing we want it to do.

The purpose of extract_frames is to pull certain frames out of an animated gif, and store them as individual image files. To do this, it wants to know several things. One is the input file, but we don't have to specify that again - actions are specific to one file at a time, so that file is automatically passed to all rules. The next thing it wants to know is the output directory to put the frame files it builds, and we assemble that using our go-to tools, bracket expressions and regex groups: `[output_root]/daily/[$1]/frame0`. [$1] still refers to the date part of the current file's path. The final thing that extract_frames wants to know is which frames to extract. In this case we simply want frame 0 of each file. The tool will create a filename for each of the frame files using a standardized naming convention it can read back later, so we don't have to specify that.

So when this runs, a Python stub calls (for the moment) the free ImageMagick tool (www.imagemagick.org), passing it the correct command-line arguments to do our frame extraction.

This rule will perform these two steps - copying the file verbatim to where we want it, and extracting out the first frame (if it's animated) for later use - for as many files as match the condition, whether it's 1 or 1000. Once the rules have run on a particular file, that file is given a marker (specifically a 'rules_run' metadata entry) so that it won't be looked at again for the rest of the run. But, we've create two new files, and now we can write more rules that look for those and do further processing (remember that the TP keeps cycling over the files until there's no more work to do). So let's make a thumbnail image with a new rule:

```
if [full] like [output_root]/daily/[date]/[any]:
    make_thumb dest: [output_root]/latest/thumb_[name], type: gif, size: 20x20
```

Notice that in this rule, we're looking for files in the output folder, not the input folder. So this will only match files we created during the last rule pass, the ones we copied into dated folders under /daily.

Our action this time is make_thumb, another plugin (which also happens to use ImageMagick) that rescales the image to a 20x20 pixel gif, and saves it to the same name as the original file but with "thumb_" prepended. (This is a thumbnail-naming convention that floe doesn't use or need, since it uses metadata instead. But it's handy for humans.)

Notice that we didn't use any parentheses in the condition regex. We didn't need to in this case, because [name] is a built-in expression for the file name (without the path) of the current file. So we could build the destination filename without match groups. We could have used them if we preferred.

One last comment on make_thumb. Because TP walks the files multiple times, rules can be written that create an infinite loop. If you wrote a naive rule matching any gif, which always

created a new gif, then TP would keep finding new files on every pass, forever. (In practice, it will abort after 20 passes because if it gets that far, something's wrong.) So, the make_thumb tool is smart: if it sees that its input file already starts with "thumb_", it refuses to make another file (which would be a thumbnail of a thumbnail, with a name starting with "thumb_thumb_". And yes, this did happen.) It's best if rules are written to avoid infinite loops anyway, but it's also nice if the tools have some common-sense precautions built in. [TODO NYI: should be using metadata rather than naming convention to detect this loop!]

**Definitions**

You can define new symbols in a rule file to simplify notation and avoid errors due to repeated text. For example, suppose you have lots of rules that reference a subfolder called 'static' off the output folder. At the top of your rule file, you can say:

```
static = [output_root]/static
```

From that point forward in the file, you can write expressions like this:

```
copy to [static]/daily/[$1]
```

Definitions can reference earlier definitions, e.g.:

```
static = [output_root]/static
images = [static]/images
```

**Rule Types**

One last thing I'll discuss about rules at this point: rule types. We've shown that rules consist of a condition followed by some actions. You can also put a header above that, like this:

```
[type: dest, label: arctic10]
if [full] like [output_root]/daily/[date]/Arctic10.gif:
    web_handle worklist: arctic10s
```

The header (as is often the case in floe) consists of key: value pairs separated by commas. `label` is just a debugging aid: you can track rule invocation and matching in the log file by setting `debug_tags` appropriately. (`RULE_MATCH_arctic10` to see when the rule matches, `RULE_ACTION_arctic10` to see when its actions file.)

The `type` value is the rule type. Every rule has a type, which determines when it runs and what set of files it sees. The default type is `self_tree`, which is the sort of rule we've seen up until now - it runs first against the input folder, then multiple times against the output folder. It's the default, so there's no need to specify it in rule headers. This rule, however, has a non-default type, `dest`. What's that mean? First of all, these rules run later in the process, after

the final set of files that are either already in the DM, or will be uploaded during this run, is known. Second, they run against that set of DM files, rather than the files in either the input or output folder.

To understand this, consider that the DM contents can be cumulative. Let's say every day we get some new files from an FTP server, process them into files we want on a website serving the DM files, and upload them. There might be a web page that shows, for example, all the differently-dated versions of Arctic10.gif that we've built since we set all this up. But we're not building all those historical files every day - that would be a huge violation of the "principle of least work" and would eventually cause a floe run to take longer than the time available. Instead, we just build today's new file, and upload it to a place where it lives with all the previous ones.

Now, to build a web page with links to all those past and present Arctic10.gif files, we need to know what they are. But we can't get that from the output folder, because that only has today's. We need to query the DM and get a list of everything that's up there (a query that it will answer using its metadata store). Then, when we run the 'dest'-type rules, we run them against that list of files, not against the output folder. That allows us to make a list of both old and new files that can be used to build a single web page that includes both.

**Worklists**

A sharp-eyed reader will have noticed an unfamiliar action on the S3 rule above:

```
web_handle worklist: arctic10s
```

What is "web_handle"? It's another plugin, but instead of doing something to the matched file, it puts it on a worklist. A worklist is a particular set of files that are going to have something done to them, or about them, later. Later, in this case, means when the WebMaker (or WM) component runs. WM is the last step in the floe pipeline, aside from cleanup tasks. As its name implies, its main job is to make all the web pages and upload them. The WM keeps track of any number of named worklists, which contain not only the names of files, but any other information about them that we might need.

So, when we are walking the tree of DM files, when we encounter an "Arctic10.gif" (old, or new), we add it to a worklist named `arctic10s`. By the end, we'll have a complete worklist of new and old Arctic10.gif files (and others too, if we made rules to add them to the list. This worklist will allow us to build a web page with a variable number of links to all those files. (Worklists have another, more advanced purpose - see "Multiple Dependencies" below.)

(Note: in a production system, we would never have a rule so specific as to only match files named "Arctic10.gif". We'd have far too many rules that way. Instead, we'd make a more general rule, and use things like match groups to get what we want.)

## Multiple Dependencies:

Everything we've looked at so far in this manual has involved taking a single new or changed file, and performing some operations on it to make new content to upload. Some floe scenarios wouldn't need anything beyond that. But some things we might want to build are the product of more than one file.

For example, one of the tools in floe can take frames that were extracted from multiple existing animations, and combine them in various ways to make a new animation. (This is being used to combine past observational data along with forecasts into one file.)

The question is, when do we need to rebuild this kind of file? If we represent it as A + B -> C (where A, B and C are files), we need to rebuild C whenever A has changed, or B has changed, or both. We really shouldn't just build C every time, because we're committed to the minimum work principle.

But the way we've set up rules doesn't lend itself to expressing, "if A is new/changed OR B is new/changed..." This could be added, but the resulting syntax would be cumbersome. Worse, since we walk the files in a particular order, and A and B might be files that are themselves built by other rules and might not both exist when the rules start running, it might be hard to know when it's safe to build C. (We don't want to build C, only to find out later that we mistakenly used the old version of B, which was replaced with a new one later in the run.)

Instead, why not just "take notes", and when we see that an A has changed, put C on a list to be rebuilt later? And when we see a changed B, do the same. As long as C gets on the "todo list" only once, we will be sure to build it only once, only when needed, and using the latest versions of A and B. The two rules would look something like this:

```
if [full] like A:
  web_handle worklist: make_c, dest_key: [output_root]/c/[$3], role: a
if [full] like B:
  web_handle worklist: make_c, dest_key: [output_root]/c/[$3], role: b
```

And the expression inside a template would look like:

```
[worklist name: make_c, mode: combine_abc]
```

I said earlier that worklists can be used for almost anything, including making new files. That's what happens here, when we encounter the template expression. The mode, "combine_abc", tells the plugin code that for every item on the worklist, we're going to combine frames from an A file and a B file into a new C file, in a particular way. (That code knows which frames to get from each file, which is easier for now than writing a general-purpose mode and specifying which

frames you want as part of the bracket expression; the general version can be added later if there's a need for it.)

There's an angle to this that we have to consider. If both A and B are new/changed, no problem, we knew that when each of their respective rules ran and put them on the worklist. But what if only A changed? B won't be in the output folder at all - there was no need to create it, as the right version is already sitting up on our website.

The solution is, we grab B from the archive. That's a critical reason for the archive to exist - it has to be there if we're going to build these multiple-dependency products. We know the B file in the archive is identical to the one on the DM (which we could go fetch, but that would be slow). It is up to the custom mode code to know where to grab any of its inputs that hasn't turned up in the output folder.

## Iterations:

In its original form, floe operated as a single-pass pipeline. So, for example, if there happened to be many gigabytes of data to download from FTP servers, that would happen first. Next, the tree-processing of all that data would run to completion, and so on. Each unit of work might be very large and take a long time to complete. This had several disadvantages: The system might run for a very long time without saving its state (particularly to the archive, at the very end of the process), meaning much lost work in the event of a crash; the user could not see whether rules, templates, and so on were working as intended without waiting a long time; and disk space and memory would have to accommodate the whole data set at once. It made sense to chop the overall task into smaller units of work that could be put through the entire pipeline in a reasonable amount of time, with all state saved before starting another chunk.

There are two config settings you can use to limit the amount of processing floe will do per iteration. This will correlate to the duration of an iteration, which will also be affected by the sizes of files and the nature of the processing you're doing to them. By section:

```
[input]
download_limit = <integer>
```

This limits how many files will be downloaded from each input manager per iteration. When it's reached, the input manager will act as if there are no more new/changed files to download, until the next iteration. This is per input manager (and each configured FTP folder has its own download manager, even if it's on the same FTP server - there is one URL DM as well, if any URLs were configured).The default is no limit.

```
[process]
root_file_limit = <integer>
```

This limits how many "root files" the TP will process per iteration. A "root file" is a file that was put in the input directory by an IM, as opposed to files that get created during the tree processing itself (either by unpacking of archives, or by rules). The reason for defining the limit on these files is that the TP will get a chance to process each such file to completion, regardless of how many passes that takes. That in turn is an attempt to leave the system in as consistent a state as possible at the end of an iteration. (This will be less of an issue if using PostgreSQL for metadata persistence.) Default: 1,000,000 files.

Note that the `include_files` and `exclude_files` entries in the `[input]` section are also key tools in the fight to control disk space and run times - eliminating work that isn't needed in the first place is even better than breaking it up into chunks.

The steps in an iteration are exactly the same as described earlier in this document. That includes template processing and upload of the resulting files. A consequence of this, which might be important for some uses, is that if the end product is a web site, it will probably be usable while the build progresses (since at the end of each iteration, a full and correct set of HTML files, in sync with the content files that have been created so far, is put in place).

By default the maximum number of iterations is 10, but for a big task you can set that higher, e.g.:

```
[actions]
max_iterations = <integer>
```

Use a large value (I use 1000) to make sure that floe runs until there's no more work to be done.

## Special modes:

Normally, floe is invoked with the path to a config file as its only argument. But there are extra arguments you can pass in addition, which cause floe to run in various special modes, while adhering to the config file settings for various essentials like paths, logging, and so forth. Most of these modes are for development, troubleshooting, or recovering from some sort of data problem without having to clean everything out and do a run from scratch.

Here are the currently supported special modes, showing the arguments to add on the command line after the config file (which is still required):

`ftp_catchup`: for each IM (FTP-based only), this reads the relevant folder on the FTP server and gets metadata (size and modified time) for each file. It then persists this data, without downloading any files. On the next regular run, floe will not see that it has any files to download.

`ftp_remove expr`: given a passed regular express string `expr`, this will remove FTP metadata for any existing files in the input folder whose full path+name matches the regex. This will cause the existing files to be invisible to floe; if on the next run the files are still on the FTP server they came from, fresh copies will be downloaded.

`ftp_rerun expr`: again, `expr` is a regex string. Any files matching will have their 'rules_run' metadata tag removed, causing floe to reprocess those files through the system on the next run. This is very useful when you want to test repeatedly without downloading files from ftp each time. (use `ftp_rerun .` to do this to every file, since '.' matches any filename.) This, despite the name, works on the URL DM as well as any FTP DMs.

`ftp_meta_from_local`: delete the existing FTP metadata and build a new set from the files currently in the input folder.

`restore_from_archive expr`: this is a way to selectively pull files back out of the archive and put them in the corresponding locations in the output folder. As usual, only files matching `expr` will be affected. You might do this so that those files can be re-uploaded to the destination, or so that rules can be run on them to produce other files. Note that you probably need to set `clear_first` to false before following this with a normal floe run - otherwise the output folder is cleared at the start of the run, deleting the files you just put there.

`test_deploy_test`: used for deployment, see below.

## Deployment:

In a production environment, files are never "built" directly to the production server. Instead, they are sent to a staging server, which is as close as possible to an identical copy of the production server. There they are subjected to a series of validation tests to make sure that something didn't "break the build", i.e. produce an unusable end product. Only after passing these tests does deployment to production occur. At this time, tests are run against the production server. If these fail, someone is notified, and in some cases the production environment is rolled back to the previous, valid state. Floe has a basic mechanism for this, the special mode `test_deploy_test`. (There are many other open-source testing and deployment tools you could use if floe's solution isn't suitable.)

This mode requires three independent destination managers to be set up: one for build, one for staging, and one for production. First, a normal build takes place, and its new files go to the build destination. The `test_deploy_test` mode then syncs the staging destination with the build destination, possibly performing "fixups" on the files.

What are fixups and why are they needed? They might not be needed, depending on the situation. Fixups are a way of doing find-and-replace operations across a folder of files. Suppose you have a configuration where image files are served from a particular URL, such as "static.mysite.com", and other files are served from plain "mysite.com". (This can be advantageous for performance, or if serving static files from a service like S3.) Links in HTML files that point to images need to reference "https://static.mysite.com/<path>". No problem.

But that's the production server. What about a staging server? Perhaps your staging server is "https://static-staging.mysite.com". To have a working staging site, your HTML links need to point to *that* instead. The solution is to build the HTML files in floe with a placeholder like _STATIC_SERVER_. Then, when copying them to the staging server, that placeholder gets replaced with "https://static-staging.mysite.com"; when copying to production, with "https://static.mysite.com". Floe has this capability (currently the static server tag _STATIC_SERVER_ is hard-coded into deploy.py; more configurable fixups will be added as needed). By the way, this explains why the build server and the staging server need to be separate. If not for fixups, they could be merged into one.

To use `test_deploy_test`, three sections are required in the config file: [build], [staging], and [production]. If not using this mode, only the [build] section is required and will be the final output destination.

The test suite is a set of URLs to request (first from the staging server, then after deployment, from the production server). Currently, the only test is for a 200 (OK) status, but this will be expanded. The URLs should be in a JSON file pointed to by the `test_urls` setting in the `[test]` section of the config file.

`test_deploy_test` will optionally accept an additional command-line parameter:

- `no_deploy` means the production server won't be modified, regardless of test results. This is useful if the automated tests need to be supplemented with other tests (from another test system, or manual). Once the staging server is verified, use `test_deploy_test` without extra parameters to deploy to production.
- `force_deploy` means content will be replicated to the production server regardless of test results. This is mostly useful during development.

## Writing plugins:

The loading of plugins, and the role of the `register` function in each plugin file, are described above. That leaves writing the code. There is no single, defined interface that plugin functions must conform to; the signature of a plugin function depends on who calls it. Plugins that are used as actions receive these arguments: `(interp, finfo, argstr)`, where:

- `interp` is either a method, or an object with a method named "interpret"; this is used in evaluating bracket expressions.

- `finfo` (for "file info") is a dictionary of information about the file that was matched - including its name, its full path or key (depending on rule type), its size and MD5 hash, and other arbitrary metadata that's been attached to the file during processing.
- `argstr` is the string of key: value pairs that were specified in the rule file after the action name.

Plugins that are called (as "modes", see above) in worklist invocation expressions are called once for each worklist item. They receive a different set of arguments: `(webmaker, dest_key, work_item, list_args)`, where:

- `webmaker` is the WM object, with which functions of this kind need to interact.
- `dest_key` is the destination key identifying the output item being processed (usually obtained by debracketing a dest_key expression in the action line of a rule).
- `work_item` is the work_item dictionary that stores information about the task. This can be simple, or more complex (e.g. in the "multiple dependencies" case described above, the work_item might contain a list of finfos of all the multiple input files that are input to some processing step). Often there is an `item_args` sub-dictionary containing other arguments that were specified in the action line.
- `list_args` is the argument string that was included in the worklist invocation expression. Both `item_args` and `list_args` can contain any keys/values that the plugin code understands, the code floe code just passes them around.

Further documentation about plugin-writing will come later.

## Future plans:

1. Support hybrid sites that use a mix of static files (generated by floe) and dynamic files. This would allow form-based tools to be built which allow site users to access the resources of floe, such as image-processing plugins; a web framework (such as Django, which is convenient because it's also Python) can load some or all of the floe code (both core and plugins) as modules. This effort is already well underway (at https://floe.keytwist.net). Per-user file areas on the server will allow large files to be processed on the server without ever being downloaded to a user's computer, and scripting will allow complex chains of processing steps to be invoked as a unit.

2. Keeping state in a real database instead of text files. (PostgreSQL is my open-source database of choice.) This will offer better performance for very large deployments, fine-grained restartability in the event of a crash, and better integration with a server-side framework (i.e. a floe installation and a Django web server can share the same PostgreSQL database server).

3. Expanded syntax for rules and actions (multiple conditions combined with AND/OR/NOT operators, bail-out on error, rules that act on the output file(s) of the preceding rule, and

others).

4. Expose floe products via REST APIs.

5. Efficiency improvements such as parallel processing, task queues, etc. This will be determined by real-world testing; no scenarios so far have demanded more performance than the current code provides.

6. More real-world use cases.

7. Better documentation.