# flo Documentation

***Release 0.2.0***

**Dean Malmgren**

August 29, 2014

Contents

`flo` is a data workflow utility that is specifically designed to enable rapid iteration and development of complex data pipelines. Its *command line interface* and *task configuration* have many features that make `flo` ideal for developing data workflows, among them:

- `flo` hashes the state of each file that it monitors to make it amenible to working with how most distributed version control systems work.
- `flo` times each step of the analysis, *making it easy to determine how long any particular run will take* before `flo` does anything.
- `flo` comes with *command line autocompletion builtin*, making it easy to evaluate your options quickly in the terminal.
- `flo`'s *task configuration is written in YAML*, making it easy to read and write without having to know an *archaic language* (sorry `make`, its not you, its me).
- `flo` is written in python, which is a native language to most data-savvy users to make it as easy as possible to maintain by the community.

If you're sold, *get started*. If not, read on:

CHAPTER 1

# op-ed

We built `flo` because *existing tools* were not cutting it for us and we kept finding ourselves saying things like:

> "It should be easy to swap in development data for production data." - @bo_p
>
> "It should be easy to work on one file at a time." - @mstringer
>
> "It should be easy to avoid making costly mistakes." - @deanmalmgren

There are many ways one could conceivably write a data analysis workflow from scratch, from writing single programs that ingest, analyze and visualize data to simple scripts that each handle one part of the puzzle. Particularly when developing workflows from scratch, we have the strong opinion that writing small scripts with intermediate outputs is a much more effective way to develop a prototype data workflow. In our experience, we find it to be very convenient to edit a script, run it, and repeat several times to make sure it is behaving the way we intend. For one thing, this pattern makes it far easier to spot check results using a litany of available command line tools. For another, this pattern makes it easy to identify weak links (*e.g.* incorrect results, poor performance, etc.) in the analysis and improve them piece by piece after the entire workflow has been written the first time.

## 1.1 design goals

This package is deliberately designed to help users write small, but compact workflow prototypes using whatever tools they prefer — R, pandas, scipy, hadoop. The goal here is not to provide a substitute for these tools, but rather to be the glue that sticks them together. It takes inspiration from a number of *existing tools* that have similar aims, particularly *GNU make* that has many desirable properties of working with data workflows. Specifically, the design goals for this project are to:

- *Provide an easy-to-use framework.* This applies for n00bs and pros alike. Use human-readable syntax.
- *Prevent, as much as reasonably possible, costly mistakes.* Avoid inadvertently rerunning commands that overwrite results or executing commands that take a long time.
- *Encourage good development practices, but allow for flexibility.* There's a tradeoff here, but we have *an opinion* on how to do this in a good way.

## 1.2 prior art

### 1.2.1 GNU make

GNU make is a very useful tool that was designed mostly for building complicated software packages. It works particularly well when you are compiling a bunch of `.c` or `.h` files into `.o` files because you can use rules to define

how `.o` files are created from a bunch of dependencies (and usually not too many of them). I've used this to manage data workflows too and, provided you're only working by yourself and you are comfortable with its arcane file format, make is a great tool. A few things make `make` less desireable, particularly for managing data workflows that tend to take hours or days rather than minutes.

- `make` uses timestamps, instead of hashes, to determine when a file is out of date with the rest. This isn't so terrible, except when you're working in an asynchronous development environment with `hg` or `git` that does not version the timestamp of files.
- `Makefiles` are extraordinarily picky and not terribly easy for n00bs to use. Even for those that are fluent in `Makefile`, looking over a Makefile is pretty cumbersome and not easy to read. This is a downer when you're trying to rapidly develop.
- `make` does not run in parallel or, if it does, it requires a deeper understanding of its arcane format than I am comfortable with. Particularly with data workflows that can potentially take days to complete, this is a very undesireable behavior.
- `make` is filesystem based, but doesn't have the ability to test whether databases or cloud storage has been updated. This is pretty important for data workflows.

### 1.2.2 invoke

Invoke is intended to be a make replacement for python with a nice Fabric-like command line interface. It is function/class based rather than file based and, because its in python, you can basically do anything you need to within an `invoke` script. Downsides include:

- Because its function/class based, there is a lot of syntactic bloat that does not make invoke scripts considerably longer than they need to be.
- Although `invoke` does provide a 'pre' keyword argument, it is not possible to run an invoke script without rerunning the *entire* workflow (git issue here). Although its certainly possible to extend invoke to address this use case, its not clear that it'll be enough to address all the use cases that we have in mind.

### 1.2.3 Fabric

Fabric is a tool that is intended for application deployment to many different servers simultaneously. It has a great command-line interface for deployment and DevOps, but doesn't provide a lot of out-of-the box functionality for managing data workflows. Downsides include:

- Fabric parallelizes tasks across machines, not tasks. In data analysis situations, you can actually divvy up analysis tasks depending on what data files are or not available.
- `fabfiles` tend to get big rather quickly, even for relatively mundane jobs. The fact that its in python is nice, but probably not necessary for running a data analysis workflow.
- There is no default way of detecting whether a task needs to be run based on timestamps or hashes. Its certainly possible to extend Fabric to address this issue.

### 1.2.4 Drake

Drake is intended to be the "make for data". `Drakefiles` have a very similar look and feel of `Makefiles`. It has some pretty decent advantages over `make` in that it comes pre-equipped with parallelization and with filesystem, S3 and HDFS integration, but there are a few key disadvantages:

- It is based on timestamps. This makes it tricky to develop when working in an asynchronous development environment like `hg` or `git`.

- Its written in clojure, which makes it difficult for most data people to contribute too (?), or at least difficult for this data person to read.

### 1.2.5 AWS Data Pipeline

Amazon's Data Pipeline is intended to organize data pipelines that occur entirely in Amazon's cloud. This seems extremely handy if you're playing entirely within Amazon's walls, but not terribly convenient for a wide range or projects where a cloud solution is unnecessarily overkill.

### 1.2.6 LONI Pipeline

Meh. http://pipeline.bmap.ucla.edu/

### 1.2.7 Predictive Modeling Markup Language (PMML)

PMML is a language to define workflows in data analysis. There appear to be many tools that will execute PMML workflows, for example Augustus and Zementis for executing on Amazon Web Services. It appears to be geared more toward developing robust, "enterprise" workflows as opposed to rapid development.

### 1.2.8 Tez

Tez appears to be the Hadoop equivalent of creating data workflows using YARN. If you're nuts about java and everything you do is in hadoop, this is probably great for you.

### 1.2.9 KNIME

KNIME is a graphical interface for defining data *and* analysis steps in a data workflow. I'm sure its possible to write custom analysis steps in KNIME to make it more practical in real world situations, but the tight coupling between the pipeline definition and actually running an analysis and doing some visualization is highly unappealing for the use cases I have in mind. Nonetheless, its worth mentioning. The GUI is admittedly kinda nice and certainly easier to understand for n00bs.

CHAPTER 2

# quick start

1. *Install this package.*

   ```
   pip install flo
   ```

2. *Write a flo.yaml.* Create a `flo.yaml` file in the root of your project. `flo.yaml` can *have many features*, but the basic idea is to make it easy to quickly define a sequence of dependent tasks in an easy-to-read way. There are several examples, the simplest of which is the hello-world example. Briefly, every task is a YAML object that has a `creates` key that represents the resource that is created by this task and a `command` key that defines the command that are required to create the resource defined in `creates`. You can optionally define a `depends` key that lists resources, either filenames on disk or other task `creates` targets, to quickly set up dependency chains.

3. *Execute your workflow.* From the same directory as the `flo.yaml` file (or any subdirectory), execute `flo run` and this will run each task defined in your `flo.yaml` until everything is complete. If any task definition in the `flo.yaml` or the contents of its dependencies change, re-running `flo run` will only redo the parts of the workflow that are out of sync since the last time you ran it. The `flo` command has *several other convenience options* to facilitate quickly writing data workflows. Running the hello-world example for the first time yields something like this:

4. *Repeat steps 2-3 until your data workflow is complete.* When developing a data workflow, it is common to write an entire workflow and then go back and revisit particular parts of the analysis. The entire purpose of this package is to make it easy to refine task definitions and quickly re-run workflows with confidence that the user will not ruin previous results or start a simulation that takes a long time.

CHAPTER 3

# flo.yaml specification

Individual analysis tasks are defined as YAML objects in a file named `flo.yaml` (or *whatever you prefer*) with something like this:

```
---
creates: "path/to/some/output/file.txt"
depends: "path/to/some/script.py"
command: "python {{depends}} > {{creates}}"
```

Every YAML object that defines a task must have *creates* and *command* keys and can optionally contain a *depends* key. The order of these keys does not matter; the above order is chosen for explanatory purposes only.

## 3.1 creates

The `creates` key uniquely identifies the resource that is created. By default, it is interpreted as a path to a file (relative paths are interpreted as relative to the `flo.yaml` file) or a directory. Importantly, every task is intended to create a single file or directory. If you have a task that creates multiple files, you can either (i) split that into separate tasks or (ii) have all of those files embedded in a directory and use the directory name as the `creates` value like this:

```
---
creates: "path/to/output/directory"
depends: "path/to/some/script.py"
command:
  - "mkdir -p {{creates}}"
  - "python {{depends}} {{creates}}"
```

In this case, the directory `path/to/output/directory` is passed as the first argument to `path/to/some/script.py`, which can then add as many files as necessary to that directory. When this task is complete, `flo` checks the hash of all files in `path/to/output/directory` and all of its child directories to determine if it is in sync or not.

## 3.2 depends

The `depends` key defines the resource(s) on which this task depends. It is common for `depends` to specify many things, including data analysis scripts or other tasks from within the `flo.yaml`. Multiple dependencies can be defined in a YAML list like this:

```
depends:
  - "path/to/some/script.py"
  - "another/task/creates/target.txt"
```

These dependencies are what `flo` uses to determine if a task is out of sync and needs to be re-executed. Importantly, `flo` obeys the dependencies when it constructs the task graph but always runs in a *deterministic order*. If a specified `depends` does not exist immediately prior to `flo` running the task, `flo` throws an informative error.

## 3.3 command

The `command` key is mandatory and it defines the command(s) that should be executed to produce the resource specified by the `creates` key. Like the `depends` key, multiple steps can be defined in a YAML list like this:

```
command:
  - "mkdir -p $(dirname {{creates}})"
  - "python {{depends}} > {{creates}}"
```

## 3.4 templating variables

Importantly, the `command` is rendered as a jinja template to avoid duplication of information that is already defined in that task. Its quite common to use `{{depends}}` and `{{creates}}` in the `command` specification, but you can also use other variables like this:

```
---
creates: "path/to/some/output/file.txt"
sigma: "2.137"
depends: "path/to/some/script.py"
command: "python {{depends}} {{sigma} > {{creates}}"
```

In the aforementioned example, `sigma` is only available when rendering the jinja template for that task. If you'd like to use `sigma` in several other tasks, you can alternatively put it in a global namespace in a flo.yaml like this (similar example here):

```
---
sigma: "2.137"
tasks:
  -
    creates: "path/to/some/output/file.txt"
    depends: "path/to/some/script.py"
    command: "python {{depends}} {{sigma} > {{creates}}"
  -
    creates: "path/to/another/output/file.txt"
    depends:
      - "path/to/another/script.py"
      - "path/to/some/output/file.txt"
    command: "python {{depends[0]}} {{sigma}} < {{depends[1]}} > {{creates}}"
```

Another common use case for global variables is when you have several tasks that all depend on the same file. You can also use jinja templating in the `creates` and `depends` attributes of your `flo.yaml` like this:

```
---
input: "data/sp500.html"
tasks:
  -
    creates: "{{input}}"
    command:
      - "mkdir -p $(dirname {{creates}})"
      - "wget http://en.wikipedia.org/wiki/List_of_S%26P_500_companies -O {{creates}}"
  -
```

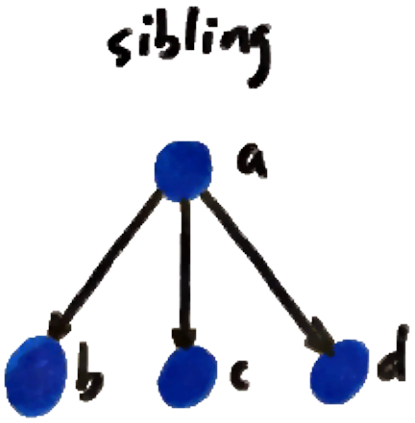
```
    creates: "data/names.dat"
    depends:
      - "src/extract_names.py"
      - "{{input}}"
    command: "python {{depends|join(' ')}} > {{creates}}"
-
    creates: "data/symbols.dat"
    depends:
      - "src/extract_symbols.py"
      - "{{input}}"
    command: "python {{depends|join(' ')}} > {{creates}}"
```

There are several examples for more inspiration on how you could use the flo.yaml specification. If you have suggestions for other ideas, please add them!

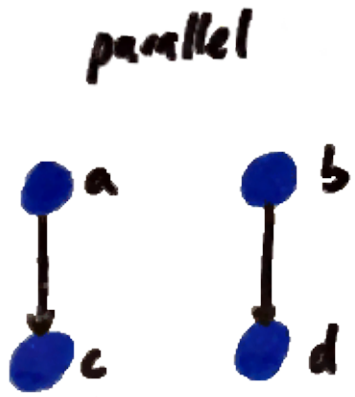
## 3.5 deterministic execution order

`flo` is *guaranteed to run in the exact same order every single time* and its important that users understand how it works. When `flo` is *executed*, it makes sure to obey the dependencies specified in the YAML configuration. In the event of ties `flo` is executed in the same order as the tasks appear in the YAML configuration. Technically, this is very similar to a breadth first search originating from the set of tasks that have no dependencies except that we order things based on the *maximum* distance that each task is from any given source node and we break ties based on the order in the YAML configuration file.

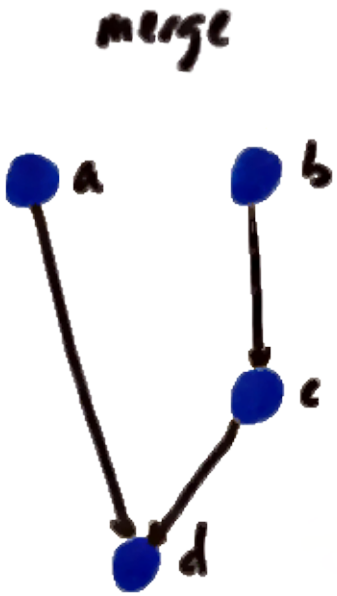
The deterministic order example contains a few different YAML configuration files to demonstrate how this works in practice, the highlights of which are summarized here.




For sibling tasks, sibling tasks are executed in the order in which they appear in the YAML configuration file, but always after the their dependencies have been satisfied. In this example, the task graph looks like this and the tasks are guaranteed to run in alphabetical order.




For parallel threads, task threads are executed based on their distance from the source tasks and secondarily based on their ordering in the YAML configuration file. In this example, the task graph looks something like this and the tasks are guaranteed to run in alphabetical order.




For merging task graphs, tasks are executed based on their maximal distance from any source task. In this example, the task graph looks something like this and the tasks are guaranteed to run in alphabetical order.

CHAPTER 4

# command line interface

This package ships with the `flo` command, which embodies the entire command line interface for this package. This command can be run from the directory that contains `flo.yaml` or any of its child directories. Output has been formatted to be as useful as possible, including the task names that are run, the commands that are run, and how long each task takes. For convenience, this information is also stored in `.flo/flo.log`.

To make the command line interface as usable as possible, autocompletion of available options with workflow is enabled by @kislyuk's amazing argcomplete package. Follow instructions to enable global autocomplete and you should be all set. As an example, this is also configured in the virtual machine provisioning for this project. Here, we elaborate on a few key features of `flo`; see `flo --help` for details about all available functionality.

## 4.1 running workflows

By default, the `flo run` command will execute the entire workflow, or at least the portion of it that is "out of sync" since the last time it ran. Executing `flo run` twice in a row without editing any files in the interim will not rerun any steps. If you edit a particular file in the workflow and re-execute `flo run`, this will only re-execute the parts that have been affected by the change. This makes it very easy to iterate quickly on data analysis problems without having to worry about re-running an arsenal commands — you only have to remember one, `flo run`.

```
flo run                # runs everything for the first time
flo run                # nothing changed; runs nothing
edit path/to/some/script.py
flo run                # only runs the parts that are affected by change
```

Importantly, if you edit a particular task in the `flo.yaml` itself, this will cause that particular task to be re-run as well:

```
flo run
edit flo.yaml          # change a particular task's command
flo run                # rerun's that command and any dependent task
```

The `flo` command is able to do this by tracking the status of all `creates`, `depends`, and task definitions by hashing the contents of these resources. If the contents in any `depends` or the task itself has changed since the last time that task was run, `flo` will run that task. For reference, the hashes of all of the `creates`, `depends`, and workflow task definitions are in `.flo/state.csv`.

## 4.2 same project, different workflows

Naturally, there will be times when you'll want to have separate sets of steps to accomplish different things. One simple way to separate your workflow configuration is by putting them in two separate files, say `figures.yaml` and `analysis.yaml`. You can then specify running these separate workflows on the command line with the `--config` option like this:

```
flo run --config figures.yaml
flo run --config analysis.yaml
```

All other behaviors of the YAML configuration and use of the `flo` command remain exactly the same.

## 4.3 limiting flo run execution

Oftentimes we do not want to run the entire workflow, but only a particular component of it. Like GNU make, you can specify a particular task by its `creates` value on the command line like this:

```
flo run path/to/some/output/file.txt
```

This limits `flo` to only executing the task defined in `path/to/some/output/file.txt` and all of its recursive upstream dependencies. Other times we do not want to run the entire workflow, but run everything after a specific task. We can do that like this:

```
flo run --start-at path/to/some/file.txt
```

This limits `flo` to only executing the task defined in `path/to/some/file.txt` and all of its recursive **downstream** dependencies. This can be combined with `flo run task_id` to only all tasks between two specified tasks like this:

```
flo run --start-at path/to/some/file.txt path/to/some/output/file.txt
```

If you ever want to only run one task, say a task that creates `path/to/some/file.txt`, you can specify that task as both the starting and ending point of the workflow run with `--only`:

```
# these two things are the same
flo run --only path/to/some/file.txt
flo run --start-at path/to/some/file.txt path/to/some/file.txt
```

In some situations — especially with very long-running tasks that you know haven't been affected by changes — it is convenient to be able to skip particular tasks like this:

```
flo run --skip path/to/some/file.txt
```

This eliminates the task associated with `path/to/some/file.txt` from the workflow but preserves the dependency chain so that other tasks are still executed in the proper order.

Sometimes it is convenient to rerun an entire workflow, regardless of the current status of the files that were generated.

```
flo run
# don't do anything for several months
echo "Rip Van Winkle awakens and wonders, where did I leave off again?"
echo "Screw it, lets just redo the entire analysis"
flo run --force
```

For long-running workflows, it is convenient to be alerted when the entire workflow completes. The `--notify` command line option makes it possible to have the last 100 lines of the `.flo/flo.log` sent to an email address specified on the command line.

```
flo run --notify j.doe@example.com
```

## 4.4 I'm nervous, what's going to happen?

While *we don't recommend it*, its not uncommon to get "in the zone" and make several edits to analysis scripts before re-running your workflow. Because we're human, its easy to incorrectly remember the files you edited and how they may affect re-running the workflow. To help, the `flo status` command lets you see which commands will be run and approximately how much time it should take (!!!).

```
flo run
edit path/to/some/script.py
edit path/to/another/script.py
echo "a long time passes"
flo status              # don't run anything, just report what would be done
```

For reference, `flo` stores the duration of each task in `.flo/duration.csv`. Another way you can comfort yourself is by looking at the status visualization.

```
flo status --serve
```

which displays something like this:

## 4.5 Starting over

Sometimes you want to start with a clean slate. Perhaps the data you originally started with is dated or you want to be confident a workflow properly runs from start to finish before inviting collaborators. Whatever the case, the `flo clean` command can be useful for removing all `creates` targets that are defined in `flo.yaml`. With the `--force` command line option, you can remove all files without having to confirm that you want to remove them. If you just want to remove a particular target, you can use `flo clean task_id` to only remove that `creates` target.

```
flo clean               # asks user if they want to remove `creates` results
flo clean --force       # removes all `creates` targets without confirmation
flo clean a/task        # only remove the a/task target
```

## 4.6 Saving results

Before removing or totally redoing an analysis, I've often found it useful to backup my results and compare the differences later. The `flo archive` command makes it easy to quickly backup an entire flo (including generated `creates` targets, source code specified in `depends`, and the underlying `flo.yaml`) and compare it to previous versions.

```
flo archive             # store archive in .flo/archives/*.tar.bz2
for i in `seq 20`; do
    edit path/to/some/script.py
    flo run
done
echo 'oh crap, this sequence of changes was a mistake'
flo archive --restore  # uncompresses archive
```

CHAPTER 5

# developing

1. Fork and clone the project:

   ```
   git clone https://github.com/YOUR-USERNAME/flo.git
   ```

2. Install Vagrant and Virtualbox and launch the development virtual machine:

   ```
   vagrant up && vagrant provision
   ```

   On `vagrant ssh`ing to the virtual machine, note that the `PYTHONPATH` and `PATH` environment variables have been altered in this virtual machine so that any changes you make to your local data workflow scripts are automatically reloaded.

3. On the virtual machine, make sure everything is working by executing workflows in `examples/*/flo.yaml`

   ```
   cd examples/reuters-tfidf
   flo run
   ```

4. To be more thorough, there is an automated suite of functional tests to make sure any patches you have made haven't disturbed the behavior of this package in any substantiative way.

   ```
   ./tests/run_functional_tests.sh
   ```

   These functional tests are designed to be run on an Ubuntu 12.04 LTS server, just like the virtual machine and the server that runs the travis-ci test suite. There are some other tests that have been added along the way in the Travis configuration. For your convenience, you can run all of these tests with:

   ```
   ./tests/run.py
   ```

   Current build status:

5. Contribute! There are several open issues that provide good places to dig in. Check out the contribution guidelines and send pull requests; your help is greatly appreciated!

CHAPTER 6

# changelog

This project uses semantic versioning to track version numbers, where backwards incompatible changes (highlighted in **bold**) bump the major version of the package.

## 6.1 latest

- enforce that `depends` must exist prior to running any commands (#59)
- more informative error messages (#56, #57, #58)
- several bug fixes, including:
  - properly handling backspacing output of subprocessed commands (#53)

## 6.2 1.0.0

- **removed pseudotask creation** (every task must have a `command` key)
- specifying alternative yaml configuration (#62)
- incorporated deterministic ordering in a predictable and explainable manner (#65, #70)
- introduced `flo status` command; **removed ‘‘flo run –dry-run‘‘ in favor of ‘‘flo status‘‘** (#55)
- incorporated the `--only` option (#37)
- several bug fixes, including:
  - making sure that the TaskGraph is a directed acyclic graph (#61)
  - ensuring that `creates` exists after a task has been run (#60)
  - clarifying output on `flo status` and `flo run` (#64)

## 6.3 0.2.0

- Initial release