flex Documentation

Release 0.1

Derek Ruths

January 25, 2016

Contents

Flex is a command-line tool for building data science pipelines, particularly in the research context. It draws its inspiration from make and linguini.

Flex seeks to address a fundamental issue that haunts every brand of data science: over the course of a project, data becomes disconnected from the processes that produced it. Once data loses its context, even the most basic interpretive tasks become exceedingly difficult. Moreover, attempts to reproduce or rebuild data become fraught as the details involved are gone or hard to reconstruct.

Flex makes it easy to create and update workflows (called pipelines) while always retaining a connection to the data that the pipeline produced.

Jump in. To quickly get up and running, check out quick_start.

CHAPTER 1

Reference materials

For more detailed information, check out the following reference materials:

1.1 Quick Start

Flex makes it easy to create and update computational workflows (called pipelines) that always retain a connection to the data that the pipeline produced.

1.1.1 Installation

The easiest way to install flex is using pypi:

```
pip install flex
```

Alternatively, install flex by downloading the source and running:

```
python setup.py install
```

or:

```
sudo python setup.py install
```

depending on permission settings for your python site-packages.

A *pipeline* is a sequence of steps (called *tasks*) that manipulates data. On a very practical level, each task takes some data as input and produces other data as output. In the example below, there are four tasks, each involved in a different part of the workflow.

```
# Pipeline: cluster_data

DATA_URL=http://foobar:8080/data.tsv
NAME_COLUMN=1
COUNT_COLUMN=2
ALPHA=2

download_data:
	code.sh:
		curl $DATA_URL > data.tsv

extract_columns: download_data
	code.sh:
```

```
                cut -f $NAME_COLUMN,$COUNT_COLUMN data.tsv | tail +2 > xdata.tsv
        code.py:
                from csv import reader
                fout = open('xdata2.tsv','w')
                for data in reader(open('xdata.tsv','r'),delimiter='\t'):
                        fout.write('%s\t%s\n' % (data[0],data[1][1:-1])

cluster_rows: extract_columns
        code.sh:
                ./cluster.sh --alpha $ALPHA xdata2.tsv > clusters.tsv

plot_clusters: cluster_rows
        code.gnuplot:
                plot "clusters.tsv" using 1:2 title 'Cluster quality'
```

Tasks can depend on other tasks (e.g., `extract_columns` depends on `download_data`) - either in the same pipeline or in other pipelines. By making tasks depend on other pipelines, it's possible to treat pipelines as modular parts of much larger workflows.

Once a task completes without error, it is *marked* - which flags it as not needing to be run again. In order to re-run a task, one can simply unmark it and run it again.

If you choose to run a task which has unmarked dependencies, these will be run before the task itself is run - in this way, an entire workflow can be run using a single command.

A task contains *blocks* which describe the actual computational steps being taken. As is seen in the example above, blocks can contain code for various different languages - making it possible to stitch together workflows that involve different languages. A single task can even contain multiple blocks for the same or different languages.

Currently, Flex supports four block types:

- ***export*** **(`export`) - this allows environment variables to be set and** unset within the context of a specific task
- *python* (`code.py`)
- *shell* (`code.sh`)
- *gnuplot* (`code.gnuplot`)

These, of course, require that the appropriate executables are present on the system. To customize the executable used, environment variables can be set (`PYTHON_EXEC` and `GNUPLOT_EXEC`, respectively).

Future releases will support additional languages natively and also provide a plugin mechanism for adding new block types.

Once a pipeline has been written, it can be run using the flex command-line tool:

```
fx run <pipeline_file>
```

The command-line tool also allows easy marking (`mark`), unmarking (`unmark`), and querying task info (`tasks`) for a pipeline.

Pipeline-specific Data

A common activity that creates a lot of data management issues is running effectively the same or similar pipelines using different parameter settings: files can get overwritten and, more generally, the user typically loses track of exactly which files came from what setting.

In Flex, files produced by a pipeline can be easily bound to their pipeline, eliminating this confusion:

```
DATA_URL=http://foobar:8080/data.tsv
ALPHA=2
...

download_data:
        code.sh:
                curl $DATA_URL > $PLN(data.tsv)

...
```

In the excerpt above, the file `data.tsv` is being bound to this pipeline using the `$PLN(.)` function. In effect, the file is prefixed (either by name or placed in a pipeline-specific directory). Future references to this file via `$PLN(data.tsv)` will access only this pipeline's version of the file - even if many pipelines are downloading the files at various times.

Extending Pipelines

In some cases, one will want to run exactly the same pipeline over and over with different parameter settings. To support this, Flex allows *extending* pipelines. Much like subclassing, extending a pipeline brings all the content of one pipeline into another one. Assume we are clustering some data using the process here (in pipeline `cluster_pln`). The process is parameterized by the alpha value.

```
# Pipeline: cluster_pln

cluster_rows: extract_columns
        code.sh:
                ./cluster.sh --alpha $ALPHA xdata2.tsv > $PLN(clusters.tsv)

plot_clusters: cluster_rows
        code.gnuplot:
                plot "$PLN(clusters.tsv)" using 1:2 title 'Cluster quality at alpha=$ALPHA'
```

We can extend this pipeline to retain the same workflow, but use different values:

```
# Pipeline: cluster_a2
extend cluster_pln

ALPHA=2
```

and again for a different value:

```
# Pipeline: cluster_a3
extend cluster_pln

ALPHA=3
```

Note that in each case the cluster data will be stored to `$PLN(clusters.tsv)`, so that each pipeline will have its own separate stored data.

Connecting Pipelines Together

It's quite reasonable to expect that one pipeline could feed into another pipeline. Flex supports this - pipelines can depend on the tasks in other pipelines - and in doing so, create even larger workflows that retain their nice modular organization.

Consider that the earlier pipeline given above, `cluster_a2`, could actually be assembling the data for a classifier. Let's break this classifier portion of the project into its own workflow:

```
# Pipeline: lda_classifier

use cluster_a2 as cdata

NEWS_ARTICLES=articles/*.gz

build_lda: cdata.cluster_rows
        export:
                LDA_CLASSIFIER=lda_runner

        code.sh:
                ${LDA_CLASSIFIER} -input $PLN(cdata,clusters.tsv) -output $PLN(lda_model.json)

label_articles: build_lda
        export:
                LDA_LABELER=/opt/bin/lda_labeler
        code.sh:
                ${LDA_LABELER} -model $PLN(lda_model.json) -data "$NEWS_ARTICLES" > $PLN(news.labels
```

In the example above, notice how the task `build_lda` both depends on a task from the `cluster_a2` pipeline and *also* uses data from that pipeline's namespace, `$PLN(cdata,clusters.tsv).`

Of course, we might want to try multiple classifiers on the same source data, so we can create other pipelines that use `cluster_a2`, shown next:

```
# Pipeline: crf_classifier

use cluster_a2 as cdata

NEWS_ARTICLES=articles/*.gz

build_crf_model: cdata.cluster_rows
        code.sh:
                /opt/bin/build_crf -data $PLN(cdata,clusters.tsv) -output $PLN(crf_model.json)

label_articles: build_crf_model
        code.py:
                import crf_model

                model = crf_model.load_model('$PLN(crf_model.json)')
                model.label_documents(docs='$NEWS_ARTICLES',out_file='$PLN(news.labels)')
```

Examples

See the `examples/` directory in the flex root directory to see some real pipelines that demonstrate the core features of the tool.

1.1.2 Command-line usage

The `fx` command provides several core capabilities:

- `fx tasks <pipeline>` will out info about one or more tasks in the pipeline including whether they are marked
- `fx run <pipeline>` will run a pipeline (or a task within a pipeline)
- `fx mark <pipeline>` will mark specific tasks or an entire pipeline

- `fx unmark <pipeline>` will unmark specific tasks or an entire pipeline

All of these commands have help messages to help their correct use.

1.2 Pipeline overview

The purposes of a pipeline are to:

- make explicit the high-level logic that achieves some goal
- connect the pipeline logic to the intermediate and final data produced

In much the same way as a make file, a flex pipeline consists of two parts:

- ***tasks*** **- these perform coherent chunks of work. In this regard, tasks are** very much the heart of the flex pipeline. In a departure from make, flex tasks themselves consist of blocks which are small language-specific units of work. For more details on this, see *Declaring an abstract pipeline*.
- ***global definitions*** **- while the tasks are the heart of flex, some** additional configurations are needed in order to make tasks easier to write and make connections between the pipeline and other pipelines. As a result, a flex pipeline starts with a global section, which allows for the definition of variables as well as for the specification of pipeline configurations and connections. For more details on this, see *Comments*.

In order to motivate the design of a pipeline, here we'll discuss two important attributes that many (most?) pipelines will have:

- they produce data
- they conceptually depend on pipelines (in a variety of ways)

These are discussed in the following subsections.

1.2.1 Data Namespacing

In order to ensure that data produced by the pipeline is always linkable to it, the pipeline maintains a namespace on the filesystem which a pipeline can easily add files to. The namespace can be one of two things:

- **a prefix that is added to any given filename. By default the prefix would** be the name of the pipeline. So if the pipeline `clean_census_info` had a task which added the file `census.tsv` to the pipeline namespace, the file would actually be named `clean_census_info_census.tsv` on disk.
- **a directory into which all files in the namespace are added. By default the** directory name is `<pipeline_name>_data`. So, given the situation above with `clean_census_info`, `census.tsv` would be written to `clean_census_info_data/census.tsv`.

The namespacing behavior can be configured in the global section, see *Configuring data prefixing*.

1.2.2 Dependencies on other pipelines

A pipeline can depend on other pipelines in two different ways.

The first and most mundane is when the tasks in one pipeline depend on the tasks in another pipeline. One can imagine that this second pipeline is *using* the first pipeline in order to build a bigger "mega" pipeline (that consists of both of them). This crops up a lot in even small-scale projects. One pipeline might deal with data download and curation, another with pre-processing data, and a third with training models or doing analysis. Each phase could be put in a separate pipeline, but they would all use one another.

The second and more subtle kind of dependency is when one pipeline is basically a modification of another pipeline. For example, suppose we want to run the same analysis using different thresholds: it's the same analysis running on the same data - with just one or two parameters set differently. Rather than duplicating all the code for the pipeline, we can create a second pipeline that *extends* the first one, just setting some specific variables to different values.

For details on these, see *Declaring dependencies*.

1.2.3 Comments

Thoroughly commenting pipelines is an important part of making them readable and maintainable. Within a flex pipeline, a comment is always one line long: beginning with a `#` symbol and continuing to the end of the line.

1.3 The global section

The global section permits the specification of configurations and variables that will affect and be available to all tasks in the pipeline.

1.3.1 Declaring variables

Note that here we offer a detailed discussion of variables within the context of the global section. For more information on variables and functions in general, see *Variable and function references*.

In keeping with UNIX shell syntax, variables are set using the syntax:

```
set <var_name>=<var_value>
```

Throughout a pipeline, the `$` character denotes a variable or function reference.

1.3.2 Configuring data prefixing

The flex system provides an easy way to create and access files and directories within the pipelines namespace. The namespace can be either a file prefix name or a directory (see *Data Namespacing* for details). The `prefix` command is used to configure this option for a given pipeline.

The general syntax for this command is:

```
prefix file/dir [prefix_path]
```

if `prefix_path` is omitted, then the following defaults are used:

- `<pipeline name>` for file prefixes
- `<pipeline name>_data` for dir prefixes

Here are some examples:

- to set the prefix to be the default file prefix, use `prefix file`
- to set the prefix to the file prefix *foobar*, use `prefix file foobar`
- to set the prefix to the default directory prefix, use `prefix dir`
- **to set the prefix to the data directory above the pipeline's** containing directory, use `prefix dir ../data`

1.3.3 Connecting other pipelines

The tasks in a single pipeline may comprise only one portion of an entire workflow. Supposing that we have a pipeline `phase1` with task `t1`, we can connect it into another pipeline using the `use` keyword in the global section.

```
use phase1

p2_task: phase1.t1
        # task stuff goes here
```

The `use` keyword also allows easier or more readable aliases to be defined:

```
use phase1 as p1

p2_task: p1.t1
        # task stuff goes here
```

1.3.4 Inheriting another pipeline

In some cases, a pipeline will be a specialization of another pipeline - it will need to use the same tasks, but perhaps define constants or parameters differently. This can often arise in machine learning contexts - different pipelines might invoke the same classifier, only with different parameters.

One way to achieve this without duplicating large sections of code is to write the shared code (tasks and variables) into one pipeline and have all the related pipelines inherit that pipeline using the `extend` keyword.

For example, suppose that we have a pipeline named `ml_master` which declares two tasks `train` and `classify` that use the value of the variable `GRID_SIZE` to build and run the classifier.

We could build a pipeline `ml_0.5` that inherits the behavior of `ml_master`, but with a specific choice of `GRID_SIZE`:

```
extend ml_master

set GRID_SIZE=0.5
```

1.3.5 Declaring an abstract pipeline

Pipelines that are meant to be extended, might not be meant to be run. This can be explicitly declared by giving the pipeline the `.afx` (abstract flex) file suffix. A pipeline declared in this way cannot be run (but any pipelines that extend it can).

1.4 The tasks section

Tasks form the heart of a pipeline: they contain the logical steps that perform actions. A single task should correspond to some meaningful and self-contained unit of work.

1.4.1 The structure of a task

Since flex is entirely concerned with capturing computational workflows, tasks contain code in executable units called *blocks*. In order to link tasks to one another, a task can depend on one or more other tasks (called its *dependencies*).

A task has the following structure:

```
<task_name>: [<dep1> <dep2> ...]
        <block1>
        <block2>
        ...
        <blockN>
```

As a simple example, here is a task named `hworld` that simply prints "Hello" followed by "world" on two separate lines:

```
hworld: other_task
        code.sh:
                MSG=Hello
                echo \$MSG
        code.py:
                msg = 'world'
                print msg
```

The task depends on another task named `other_task`. In order to print the results, it uses two code blocks - one containing a shell script and one containing a python script. The details of the syntax here will be discussed in the following section.

1.4.2 Declaring dependencies

A dependency is another task. To declare a dependency, simply put the task name in the task declaration line after the colon:

```
# first_pipeline

first:
        code.sh:
                echo 'first'

second: first
        code.sh:
                echo 'second'
```

In the example above, the task `second` has one dependency: `first`.

In situations where a pipeline has been included with the `use` keyword, tasks in the included pipeline can be dependencies. To do this, use `<pipeline_name>.<task_name>` to refer to the task. If an alias was given for the pipeline, then the alias must be used:

```
use first_pipeline as fp

third: fp.second
        code.sh:
                echo 'third'
```

1.4.3 Declaring blocks

A block corresponds to a unit of executable code *in a specific language*. A single block might be written in bash, python, or any other supported language.

A block consists of the block declaration line (indented one tab) followed by the block contents (all of which is intended two levels).

Block declaration. The block declaration line indicates what language is being used. `code.sh` corresponds to the shell language, `code.py` corresponds to python. Currently the following languages are supported:

- Bash - `code.sh`
- Python - `code.py`
- Gnuplot - `code.gpl`
- Awk - `code.awk`

There is also another special block called `export` which accepts variable declarations using the same format as the globals section. *export* blocks can be used to set variables within the scope of this specific task.

Block content. Block content is further indented under the block declaration line. For example:

```
task1:
        code.sh:
                ls -1 > contents.txt
        code.py:
                x = 1
                y = 2
                print 'Two numbers: %d %d' % (x,y)
```

In this example, there are two code blocks. The contents of the code block can contain arbitrary content that adheres to the language of the block.

Execution order. When a task contains more than one block, the blocks are executed in the order in which they are declared in the pipeline file. So in the example above, the shell block would be executed, followed by the python block.

Use of variables. Variables and functions will be discussed in much more detail in *Variable and function references*. While discussing blocks, however, several points are worth noting.

Before the block content is passed to the appropriate execution system (e.g., the python interpreter), flex variables and functions are first evaluated. All variables and functions begin with a `$` character:

```
# var_test pipeline
in_dir=/etc
out_fname=output.dat

do_it:
        export:
                tmp_file=__foobar.txt
        code.sh:
                export PATH=~/local/bin:\$PATH
                ls -l $in_dir > $tmp_file
                cut -f1 > $out_fname
```

In the example above, the shell code block makes use of three flex-defined variables, `in_dir`, `tmp_file`, and `out_fname`. Notice that it also references the shell variable `PATH` and that, in order to make this reference, a backslash is used to escape the `$` character.

Configuring the execution environment. All flex variables are exported into the shell environment in which the execution system will run. For example:

```
PYTHONPATH=.

do_it:
        code.py:
                import mylib
                mylib.run()
```

sets the *PYTHONPATH* variable that the python interpreter will use.

1.4.4 Overloading tasks

Situations can arise in which a pipeline is extending another pipeline, but wants a particular task to do something different. This task *overloading* is achieved simply by defining the task again in the current pipeline:

```
extend first_pipeline

first:
        code.py:
                print 'this is the first task'
```

In this case, we have overloaded *first* task from earlier to print out a different message.

1.5 Variable and function references

As alluded to in earlier sections, variables and functions are important to writing modular, readable, and maintainable pipelines. Here we discuss the guts of how variables, variable references, and function invocations are handled and resolved.

1.5.1 Syntax

Much like in bash and make, variables and functions are references using `$<name>` or `${<name>}`, where the name is the name of the variable or function. Functions have the additional requirement of parentheses which contain the input arguments: `$<fxn_name>(<args>)` or `${<fxn_name>}(<args>)`.

Variable and function names can consist of one or more alphanumeric or underscore characters. The second reference form using curly braces allows the use of variables in places where there is no whitespace: `foobar_${iternum}.txt`.

1.5.2 Available functions

Executing shell commands

The `$(x)` command executes command `x` and evaluates to the standard out produced by the execution. To be valid, the command must produce exactly one line of text.

```
cmd = gcc
t1:
        code.sh:
                ls -lh $(which $cmd)
```

In this example, the which command is run. Notice that flex variables can be used within functions.

Accessing resources in the namespace

The `$PLN(x)` function will resolve to the absolute path to the resource `x` within the pipeline namespace. So, if the pipeline namespace is `/tmp/foobar`, then `$PLN(x) = /tmp/foobar_x`.

Accessing resources in ‘other‘ namespaces. At times it may be necessary for one pipeline to access a resource in another pipeline’s namespace. The `$PLN(p,r)` function can be used for this purpose. Here the function accepts two arguments. `p` is the name of the pipeline (which must be mentioned in a `use` statement) and `r` is the resource name. For example, in the following code:

```
use phase1 as p1

p2_task: p1.t1
        code.sh:
                head $PLN(p1,foobar.txt)
```

`p2_task` will access the file `foobar.txt` in the namespace of the phase1 pipeline.

1.5.3 Resolution rules

Variable and function references are resolved in two places:

- The right-hand side of variable assignments
- Anywhere inside blocks

Consider the following example pipeline:

```
my_site_packages=$(which python | basedir)/lib/site-packages

iter_num=10

download:
        export:
                PYTHONPATH=$my_site_packages
        code.py:
                import mylib
                mylib.run($iter_num)
```

In it, a number of flex variables and functions are used. Notably, the reference to `$iter_num` is resolved to `10` before the python code is called.

1.5.4 Global vs. block scope

Any variables defined in or changed in *export* blocks do not retain those affects outside of the task in which they appear.

1.6 The `fx` command

All of flex's functionality is accessed through the `fx` command line tool. You'll first need a pipeline, of course. For illustration purposes, throughout this section, we'll use the pipeline `foobar`, which has the following contents:

```
use configurator as config

download_data: config.setup_env
        code.sh:
                curl www.greatdata.com/dataset1.tsv > $PLN(dataset1.tsv)

extract_col1: download_data
        code.sh:
                cut -f 1 $PLN(dataset1.tsv) > $PLN(col1.txt)
```

A bit of vocabulary will help our discussion of the behavior of the `fx` command:

- **a *direct dependency of a task* (say, `taskX`) is another task which** appears in the dependency list of `taskX`. In `foobar`, `download_data` is direct dependency of `extract_col1`.
- **the *dependencies of a task* (say, `taskX`) are *all* the direct** dependencies of `taskX` as well as the direct dependencies of those tasks and so forth. In `foobar`, the dependencies of `extract_col1` includes `download_data` as well as `config.setup_env` and an tasks that `setup_env` depends on.
- **a *terminal task* is a task that isn't in the dependency list of any other** task in the pipeline. In `foobar`, `extract_col1` is the only terminal task.
- **a task becomes *marked* when it is successfully run. Typically this is used** to ensure that the task isn't run again.

1.6.1 Running a pipeline

The most fundamental activity we'll need to do is running a tasks in a pipeline.

Running a complete pipeline. To run all tasks in your pipeline, use `fx run <pipeline_file>`. This will run all unmarked terminal tasks and their dependencies. They are run in dependency order - so the terminal task will be the last task run. For details on the rules that govern if and when a dependency is run, see *When and if dependencies are run*.

Running a specific task. To run a specific task in your pipeline, use `fx run <pipeline_file> <task_name>`. This will run the task (if unmarked) as well as its dependencies.

Running marked tasks. If you do want to run a task that has already been marked, you have two options.

1. Unmark the relevant task using the `fx unmark` command.

2. Use the `-f` flag to force tasks to be run. This flag takes an argument which determines what tasks are forced to run.

   - `run -f=NONE` doesn't override any markings. This is the default

   behavior.

   - `run -f=TOP` overrides the marking on only the terminal task/specified

   task.

   - `run -f=ALL` overrides the markings on all tasks encountered during the

   run. *Be careful* when using this option as it can cause tasks far down the dependency tree to be re-run.

   - `run -f=SOLO` ignores any dependencies the named task may have and runs just that task.

When and if dependencies are run

By default, when flex wants to run a task (we'll call this the *final task*), it will first check to see if any of the dependencies of that task need to be run first.

The order of dependency evaluation is set such that a particular task is never evaluated before its direct dependencies. When this policy is applied to all dependencies of the final task, we end with an ordering that starts with the tasks which have no dependencies and end with the final task.

When a task is being evaluated, it is run if either of the following conditions are true:

1. the task is unmarked
2. the task's mark is older than one of its direct dependencies.

In either of these cases, the task will be run and, if successful, it will be marked.

1.6.2 Marking and unmarking tasks

To mark a specific task, use `fx mark <pipeline_file> <task_name>`. If the task specified is not marked, it will be marked. If the task is already marked, then the timestamp on the task's mark will be updated.

To unmark a specific task, use `fx unmark <pipeline_file> <task_name>`. This will remove the mark on the task (if it exists).

1.6.3 Checking status of pipeline tasks

You can use the `fx tasks <pipeline_file>` command to print out information about all the tasks in the pipeline. This will print the tasks in the pipeline as well as any tasks in other pipelines on which it depends. The timestamp of any marked tasks will be given.

1.7 Tips and Tricks

Flex is a too young to have conventions, per say. But there are some tips and tricks that can be useful.

1.7.1 Pipeline naming

Use nouns. Name pipelines for the literal thing they're doing. For example, a pipeline that obtains and prepares data from the US census might be called `us_census_grabber`.

No suffixes. Like makefiles, pipelines should be named without a suffix. Since the default namespace is based off of the pipeline's filename, this avoids ugly file and directory names.

Abstract pipelines. Abstract pipelines should be named in such a way that it is clear that they contain placeholders. If the pipeline is designed to model a particular country, then the name might be `XX_model_builder` where `XX` is a stand-in for the country code (which will be specified in the derived pipelines.

1.7.2 Extending pipelines

When to extend. When you have a certain kind of analysis that you'd like to run on different datasets or using different parameter values, write an abstract pipeline.

Don't make extended pipelines functional. In order to avoid confusing the purpose of an abstract pipeline with those that actually do work, avoid running a pipeline that will be extended.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search