pipelines Documentation

Release 0.9.0-dev

Bock lab

Apr 07, 2018

Getting started

Looper is a python application that deploys pipelines across samples with minimal effort. To get started, proceed with the *Introduction*. If you're looking for actual pipelines, you can find a list in the Hello, Looper! example repository.

CHAPTER 1

Contents

1.1 Introduction

Looper is a pipeline submitting engine. Once you've built a command-line pipeline, Looper helps you deploy that pipeline across lots of samples. Looper standardizes the way the user (you) communicates with pipelines. While most pipelines specify a unique interface, looper lets you to use the same interface for every pipeline and every project. As you have more projects, this will save you time.

Looper is scalable: For simple jobs, it can run pipelines sequentially on the local computer. For larger needs, a simple change switches to creating and submitting jobs to any cluster resource manager (like SLURM, SGE, or LFS). It's totally configurable. We provide sensible defaults for ease-of-use, but you can configure just about anything.

Looper reads projects that follow the standardized Portable Encapsulated Project format. This simple format uses a *yaml* configuration file plus a *csv* sample annotation file. Looper reads this metadata and submits pipeline runs for each sample.

Looper makes it easy to:

- only submit jobs that haven't already been submitted
- run multiple pipelines on each sample
- interface with any kind of pipeline
- collate inputs from different locations on disk
- request different resources for different input file sizes
- monitor which jobs are running or failed
- run different pipelines on different types of data
- use sample objects for downstream (post-pipeline) data analysis

1.1.1 How does it work?

We provide a format specification (the *project config file*), which you use to describe your project. You create this single configuration file (in yaml format), pass this file as input to `looper`, which parses it, reads your sample list, maps each sample to the appropriate pipeline, and creates and runs (or submits) job scripts.

1.1.2 What looper does NOT do

Looper **is not a pipeline workflow engine**, which is used to build pipelines. Looper assumes you already have pipelines built; if you're looking to build a new pipeline, we recommend pypiper, but you can use looper with **any pipeline that accepts command-line arguments**. Looper will then map samples to pipelines for you.

1.1.3 Why should I use looper?

The philosophical rationale for looper is that it **decouples sample handling from pipeline processing**. This creates a modular system that subscribes to the unix philosophy, which provides many benefits. In many published bioinformatics pipelines, sample handling (submitting different samples to a cluster) is delicately intertwined with pipeline commands (running the actual code on a single sample). Often, it is impossible to divide sample handling from the pipeline itself. This approach leads to several challenges that can be reduced by separating the two:

- running a pipeline on just a few samples or just a single test case for debugging may require a full-blown distributed compute environment if the system is not set up to work locally.
- pipelines that handle multiple samples must necessary implement sample handling code, which in theory could be shared across many pipelines. Instead, most pipelines re-implement this, leading to a unique (and often sub-par) sample handling system for each published pipeline.
- if each pipeline has its own sample processing code, then each also necessarily must define a unique interface: the expected folder structure, file naming scheme, and sample annotation format. This makes it nontrivial to move a dataset from one pipeline to another.

The modular approach taken by looper addresses these issues. By dividing sample processing from pipelining, the sample processing code needs only be written once (and can thus be written well) – that's what looper is. The user interface can be made simple and intuitive, and a user must then learn only a single interface, which will work with any pipeline.

1.2 Features and benefits

Simplicity for the beginning, power when you need to expand.

- **Flexible pipelines:** Use looper with any pipeline, any library, in any domain. We designed it to work with pypiper, but looper has an infinitely flexible command-line argument system that will let you configure it to work with any script (pipeline) that accepts command-line arguments. You can also configure looper to submit multiple pipelines per sample.
- **Flexible compute:** If you don't change any settings, looper will simply run your jobs serially. But Looper includes a templating system that will let you process your pipelines on any cluster resource manager (SLURM, SGE, etc.). We include default templates for SLURM and SGE, but it's easy to add your own as well. Looper also gives you a way to determine which compute queue/partition to submit on-the-fly, by passing the `--compute` parameter to your call to `looper run`, making it simple to use by default, but very flexible if you have complex resource needs.

- **Standardized project definition:** Looper reads a flexible standard format for describing projects, which we call PEP (Portable Encapsulated Projects). Once you describe your project in this format, other PEP-compatible tools can also read your project. For example, you may use the pepr R package or the (pending) `pep` python package to import all your sample metadata (and pipeline results) in an R or python analysis environment. With a standardized project definition, the possibilities are endless.
- **Subprojects:** Subprojects make it easy to define two very similar projects without duplicating project metadata.
- **Job completion monitoring:** Looper is job-aware and will not submit new jobs for samples that are already running or finished, making it easy to add new samples to existing projects, or re-run failed samples.
- **Flexible input files:** Looper's *derived column* feature makes projects portable. You can use it to collate samples with input files on different file systems or from different projects, with different naming conventions. How it works: you specify a variable filepath like `/path/to/{sample_name}.txt`, and looper populates these file paths on the fly using metadata from your sample sheet. This makes it easy to share projects across compute environments or individuals without having to change sample annotations to point at different places.
- **Flexible resources:** Looper has an easy-to-use resource requesting scheme. With a few lines to define CPU, memory, clock time, or anything else, pipeline authors can specify different computational resources depending on the size of the input sample and pipeline to run. Or, just use a default if you don't want to mess with setup.

1.3 Installing and Hello, World!

Release versions are posted on the GitHub looper releases page. You can install the latest release directly from GitHub using pip:

```
pip install --user https://github.com/pepkit/looper/zipball/master
```

Update looper with:

```
pip install --user --upgrade https://github.com/pepkit/looper/zipball/master
```

To put the `looper` executable in your `$PATH`, add the following line to your `.bashrc` or `.profile`:

```
export PATH=~/.local/bin:$PATH
```

Now, to test looper, follow the commands in the Hello, Looper! example repository. Details are located in the README file; Briefly, just run these 5 lines of code and you're running your first looper project!

```
# Install the latest version of looper:
pip install --user https://github.com/pepkit/looper/zipball/master

# download and unzip this repository
wget https://github.com/databio/hello_looper/archive/master.zip
unzip master.zip

# Run it:
cd hello_looper-master
looper run project_config.yaml
```

Hint: If the looper executable isn't in your path, add it with `export PATH=~/.local/bin:$PATH`.

Now just read the explanation in the Hello, Looper! example repository to understand what you've accomplished.

1.4 Extended tutorial

The best way to learn is by example, so here's an extended tutorial to get you started using looper to run pre-made pipelines on a pre-made project.

First, install looper and pypiper. Pypiper is our pipeline development framework; it is not required to use looper, which can work with any command-line pipeline, but this tutorial uses pypiper pipelines so we must install it now:

```
pip install --user https://github.com/pepkit/looper/zipball/master
pip install --user https://github.com/databio/pypiper/zipball/master
```

Now, you will need to grab a project to run, and some pipelines to run on it. We have a functional working project example and an open source pipeline repository on github.

```
git clone https://github.com/epigen/microtest.git
git clone https://github.com/epigen/open_pipelines.git
```

Now you can run this project with looper! Just use `looper run`:

```
looper run microtest/config/microtest_config.tutorial.yaml
```

Hint: If the looper executable isn't in your path, add it with `export PATH=~/.local/bin:$PATH`.

1.4.1 Pipeline outputs

Outputs of pipeline runs will be under the directory specified in the `output_dir` variable under the `paths` section in the project config file (see *Configuration files*) this is usually the name of the project being run.

Inside there will be two directories:

- `results_pipeline` - pipeline results with one subdirectory per sample
- `submissions` - yaml representations of the samples and log files of the submitted jobs.

In this example, we just ran one example sample (an amplicon sequencing library) through a pipeline that processes amplicon data (to determine percentage of indels in amplicon).

From here to running hundreds of samples of various sample types is virtually the same effort!

1.4.2 On your own

To use looper on your own, you will need to prepare 2 things: your project (what data do you want to process), and your pipelines (what do you want to do with that data). The next sections provide detailed instructions on how to tell looper about these 2 things:

1. **Project**. To link your project to looper, you will need to *define your project* using looper's standard format.
2. **Pipelines**. You will want to either use pre-made looper-compatible pipelines, or link your own, custom built pipelines. Either way, the next section includes detailed instructions on how to *connect your project to your pipeline*.

1.5 How to define a project

`Looper` subscribes to standard PEP format. So to use `looper`, you first define your project using Portable Encapsulated Project (PEP) structure. PEP is a standardized way to represent metadata about your project and each of its samples. If you follow this format, then your project can be read not only by `looper`, but also by other software, like the pepr R package, or the peppy python package. This will let you use the same metadata description for downstream data analysis.

The PEP format is simple and modular and uses 2 key files:

1. **Project config file** - a `yaml` file describing file paths and project settings
2. **Sample annotation sheet** - a `csv` file with 1 row per sample

You can find complete details of both files at the official documentation for Portable Encapsulated Projects.

Once you've specified a PEP, it's time to link it to the looper pipelines you want to use with the project. You'll do this by adding one more line to your project config file to indicate the **pipeline_interfaces** you need. This is described in the next section on how to *link a project to a pipeline*).

1.6 How to link a project to a pipeline

One of the advantages of looper is that it decouples projects and pipelines, so you can have many projects that all use the same pipeline, or many pipelines running on the same project. This modular connection between pipelines and projects happens through a file called the *pipeline interface*.

If you're using one or more existing looper-compatible pipelines, all you have to do is point your project config file at the *pipeline interface* files for any pipelines your project needs. For most casual users of pipelines, that's all you'll need to do; you'll never need to create a new *pipeline interface* file.

But **if you need to make a new pipeline looper-compatible**, you do this by creating a *pipeline interface* file for the pipeline. This lets the pipeline author tell looper how to run the pipeline. This is explained in the next section, *Writing a pipeline interface*.

Many projects will require only existing pipelines that are already looper-compatible. We maintain a (growing) list of public looper-compatible pipelines that will get you started. This list includes pipelines for data types like RNA-seq, bisulfite sequencing, etc.

To use one of these pipelines, just clone the pipeline repository add the path to the pipeline's *pipeline_interface.yaml* file to the *pipeline_interfaces* attribute in the *metadata* section of your *project_config* file:

```
metadata:
  pipeline_interfaces: /path/to/pipeline_interface.yaml
```

This value should be the absolute path to the *pipeline interface* file (wherever you cloned the pipeline repository). After that, you just need to make sure your project definition provides all the necessary sample metadata required by the pipeline. For example, you will need to make sure your sample annotation sheet specifies the correct value under *protocol* that your linked pipeline understands. These details are specific to each pipeline and should be defined in the pipeline's README.

You can also **link more than one pipeline** to your project by simply adding other *pipeline interface* files to a list in the *metadata.pipeline_interfaces* field, like this:

```
metadata:
  pipeline_interfaces: [/path/to/pipeline_interface1.yaml, /path/to/pipeline_
↪interface2.yaml]
```

For more details, see *How to link to multiple pipelines*.

1.7 How to link to multiple pipelines

If you have a project that contains samples of different types, then you may need to **link more than one pipeline** to your project. You do this by simply adding other *pipeline interface* files to a list in the *metadata.pipeline_interfaces* field, like this:

```
metadata:
  pipeline_interfaces: [/path/pipeline_interface1.yaml, /path/pipeline_interface2.
↪yaml]
```

In this case, for a given sample, looper will first look in `pipeline_interface1.yaml` to see if appropriate pipeline exists for this sample type. If it finds one, it will use this pipeline (or set of pipelines, as specified in the `protocol_mappings` section of the `pipeline_interface.yaml` file). Having submitted a suitable pipeline it will ignore the pipeline_interface2.yaml interface. However if there is no suitable pipeline in the first interface, looper will check the second and, if it finds a match, will submit that. If no suitable pipelines are found in any of the interfaces, the sample will be skipped as usual.

If your project contains samples with different protocols, you can use this to run several different pipelines. For example, if you have ATAC-seq, RNA-seq, and ChIP-seq samples in your project, you may want to include a *pipeline interface* for 3 different pipelines, each accepting one of those protocols. In the event that more than one of the *pipeline interface* files provide pipelines for the same protocol, looper will only submit the pipeline from the first interface. Thus, this list specifies a *priority order* to pipeline repositories.

1.8 How to write a pipeline interface

Hint: If you're just a pipeline **user**, you don't need to worry about this section. This is for those who want to configure a new pipeline or an existing pipeline that isn't already looper-compatible.

Looper can connect samples to any pipeline, as long as it runs on the command line and uses text command-line arguments. These pipelines could be simple shell scripts, python scripts, perl scripts, or even pipelines built using a framework. Typically, we use python pipelines built using the pypiper package, which provides some additional power to looper, but this is optional.

Regardless of what pipelines you use, you will need to tell looper how to interface with your pipeline. You do that by specifying a **pipeline interface file**. The **pipeline interface** is a `yaml` file with two subsections:

1. `protocol_mapping` - maps sample `protocol` (aka `library`) to one or more pipeline scripts.
2. `pipelines` - describes the arguments and resources required by each pipeline script.

Let's start with a very simple example. A basic `pipeline_interface.yaml` file may look like this:

```
protocol_mapping:
  RRBS: rrbs_pipeline

pipelines:
  rrbs_pipeline:
    name: RRBS
    path: path/to/rrbs.py
    arguments:
      "--sample-name": sample_name
      "--input": data_path
```

The first section specifies that samples of protocol `RRBS` will be mapped to the pipeline specified by key `rrbs_pipeline`. The second section describes where the pipeline with key `rrbs_pipeline` is located and what command-line arguments it requires. Pretty simple. Let's go through these 2 sections in more detail:

1.8.1 Pipeline interface section: protocol_mapping

The `protocol_mapping` section explains how looper should map from a sample protocol (like `RNA-seq`, which is a column in your annotation sheet) to a particular pipeline (like `rnaseq.py`), or group of pipelines. Here's how to build `protocol_mapping`:

- **Case 1:** one protocol maps to one pipeline. Example: `RNA-seq: rnaseq.py`

Any samples that list "RNA-seq" under `library` will be run using the `rnaseq.py` pipeline. You can list as many library types as you like in the protocol mapping, mapping to as many pipelines as you configure in your `pipelines` section.

Example `protocol_mapping` section:

```
RRBS: rrbs.py
WGBS: wgbs.py
EG: wgbs.py
ATAC: atacseq.py
ATAC-SEQ: atacseq.py
CHIP: chipseq.py
CHIP-SEQ: chipseq.py
CHIPMENTATION: chipseq.py
STARR: starrseq.py
STARR-SEQ: starrseq.py
```

- **Case 2:** one protocol maps to multiple independent pipelines. Example: `Drop-seq: quality_control.py, dropseq.py`

You can map multiple pipelines to a single protocol if you want samples of a type to kick off more than one pipeline run.

Example `protocol_mapping` section:

```
SMART-seq:  >
  rnaBitSeq.py -f,
  rnaTopHat.py -f
```

1.8.2 Pipeline interface section: pipelines

The `pipelines` section defines important information about each pipeline, including its name, script location on disk, and optional or required command-line arguments. In addition, if you're using a cluster resource manager, it also specifies which compute resources to request. For each pipeline, you specify values for a few specific keys. Let's start with a basic example:

```
pipelines:
  pipeline_key:  # this is variable (script filename)
    name: pipeline_name  # used for assessing pipeline flags (optional)
    path: relative/path/to/pipeline_script.py
    looper_args: True
    arguments:
      "-k" : value
      "--key2" : value
```

```
    "--key3" : null # value-less argument flags
  resources:
    default:
      file_size: "0"
      cores: "4"
      mem: "6000"
      time: "2-00:00:00"
    resource_package_name:
      file_size: "2"
      cores: "4"
      mem: "6000"
      time: "2-00:00:00"
```

Each pipeline gets its own section (here there's just one, `pipeline_key`). The particular keys that you may specify for each pipeline are:

- `name` (recommended): Name of the pipeline. This is used to assess pipeline flags (if your pipeline employs them, like pypiper pipelines).
- `path` (required): Absolute or relative path to the script for this pipeline. Relative paths are considered **relative to your pipeline_interface file**. We strongly recommend using relative paths where possible to keep your pipeline interface file portable. You may also use shell environment variables (like `${HOME}`) in the `path`.
- `arguments` (required): List of key-value pairs of arguments required by the pipeline. The key corresponds verbatim to the string that will be passed on the command line to the pipeline, that is, the absolute, quoted name of the argument (like `"--input"`). The value corresponds to an attribute of the sample, which will be derived from the sample_annotation csv file (in other words, it's a column name of your sample annotation sheet). Looper will find the value of this attribute for eash sample and pass that to the pipeline as the value for that argument. For flag-like arguments that lack a value, you may specify *null* as the value (e.g. *"–quiet-mode": null*). These arguments are considered **required**, and looper will not submit a pipeline if a sample lacks an attribute that is specified as a value for an argument.
- `optional_arguments`: Any arguments listed in this section will be passed to the pipeline *if the specified attribute exists for the sample*. These are considered optional, and so the pipeline will still be submitted if they are not provided.
- `required_input_files` (optional): A list of sample attributes (annotation sheets column names) that will point to input files that must exist.
- `all_input_files` (optional): A list of sample attributes (annotation sheet column names) that will point to input files that are not required, but if they exist, should be counted in the total size calculation for requesting resources.
- `ngs_input_files` (optional): For pipelines using sequencing data, provide a list of sample attributes (annotation sheet column names) that will point to input files to be used for automatic detection of `read_length` and `read_type` sample attributes.
- `looper_args` (optional): Provide `True` or `False` to specify if this pipeline understands looper args, which are then automatically added for
  - `-C`: config_file (the pipeline config file specified in the project config file; or the default config file, if it exists)
  - `-P`: cores (the number of cores specified by the resource package chosen)
  - `-M`: mem (the memory limit)
- `resources` (recommended): A section outlining how much memory, CPU, and clock time to request, modulated by input file size. If the `resources` section is missing, looper will only be able to run the pipeline locally (not submit it to a cluster resource manager). If you provide a `resources` section, you must define at

least 1 option named 'default' with `file_size: "0"`. Then, you define as many more resource "package" as you want. The `resources` section can be a bit confusing. Think of it like a group of steps of increasing size. The first step (default) starts at 0, and this step will catch any files that aren't big enough to get to the next level. Each successive step is larger. Looper determines the size of your input file, and then iterates over the resource packages until it can't go any further; that is, the `file_size` of the package is bigger (in gigabytes) than the input file size of the sample. At this point, iteration stops and looper has selected the best-fit resource package for that sample – the smallest package that is still big enough. Add as many additional resource sets as you want, with any names. Looper will determine which resource package to use based on the `file_size` of the input file. It will select the lowest resource package whose `file_size` attribute does not exceed the size of the input file. Becuase the partition or queue name is relative to your environment, we don't usually specify this in the `resources` section, but rather, in the `pepenv` config. So, *file_size: "5"* means 5 GB. This means that resource package only will be used if the input files total size is greater than 5 GB.

Here is a more complete example that includes multiple pipelines:

```
pipelines:
  rrbs:
    name: RRBS
    looper_args: True
    path: path/to/rrbs.py
    arguments:
      "--sample-name": sample_name
      "--genome": genome
      "--input": data_path
      "--single-or-paired": read_type
    resources:
      default:
        file_size: "0"
        cores: "4"
        mem: "4000"
        time: "2-00:00:00"
      high:
        file_size: "4"
        cores: "6"
        mem: "4000"
        time: "2-00:00:00"

  rnaBitSeq.py:
    looper_args: True
    arguments:
      "--sample-name": sample_name
      "--genome": transcriptome
      "--input": data_path
      "--single-or-paired": read_type
    resources:
      default:
        file_size: "0"
        cores: "6"
        mem: "6000"
        time: "2-00:00:00"

  atacseq.py:
    arguments:
      "--sample-yaml": yaml_file
      "-I": sample_name
      "-G": genome
    looper_args: True
    resources:
```

```
    default:
      file_size: "0"
      cores: "4"
      mem: "8000"
      time: "08:00:00"
```

1.9 How to run jobs on a cluster

By default, looper will build a shell script for each sample and then run each sample serially on the local computer. But where looper really excels is in large projects that require submitting these jobs to a cluster resource manager (like SLURM, SGE, LFS, etc.). Looper handles the interface to the resource manager so that projects and pipelines can be moved to different environments with ease.

To configure looper to use cluster computing, all you have to do is tell looper a few things about your cluster setup: you create a configuration file (*compute_config.yaml*) and point an environment variable (`PEPENV`) to this file, and that's it!

Following is a brief overview to familiarize you with how this will work. When you're ready to hook looper up to your compute cluster, you should follow the complete, step-by-step instructions and examples in the pepenv repository at https://github.com/pepkit/pepenv.

1.9.1 PEPENV overview

Here is an example `compute_config.yaml` file that works with a SLURM environment:

```
compute:
  default:
    submission_template: templates/local_template.sub
    submission_command: sh
  local:
    submission_template: templates/local_template.sub
    submission_command: sh
  slurm:
    submission_template: templates/slurm_template.sub
    submission_command: sbatch
    partition: queue_name
```

The sub-sections below `compute` each define a "compute package" that can be activated. By default, the package named `default` will be used, which in this case is identical to the `local` package. You can make your default whatever you like. You may then choose a different compute package on the fly by specifying the `--compute` argument to looper run like so: `looper run --compute PACKAGE`. In this case, `PACKAGE` could be either `local` (which would do the same thing as the default, so doesn't change anything) or `slurm`, which would run the jobs on slurm, with queue `queue_name`. You can make as many compute packages as you wish (for example, to submit to different slurm partitions).

There are two or three sub-parameters for a compute package:

- **submission_template** is a (relative or absolute) path to the template submission script. Templates files contain variables that are populated with values for each job you submit. More details are in the pepenv readme.
- **submission_command** is the command-line command that *looper* will prepend to the path of the produced submission script to actually run it (*sbatch* for SLURM, *qsub* for SGE, *sh* for localhost, etc).
- **partition** specifies a queue name (optional).

1.9.2 Resources

You may notice that the compute config file does not specify resources to request (like memory, CPUs, or time). Yet, these are required as well in order to submit a job to a cluster. In the looper system, **resources are not handled by the pepenv file** because they not relative to a particular computing environment; instead they are are variable and specific to a pipeline and a sample. As such, these items are defined in the `pipeline_interface.yaml` file (`pipelines` section) that connects looper to a pipeline. The reason for this is that the pipeline developer is the most likely to know what sort of resources her pipeline requires, so she is in the best position to define the resources requested.

For more information on how to adjust resources, see the *pipeline interface* documentation. If all the different configuration files seem confusing, now would be a good time to review *who's who in configuration files*.

1.10 How to handle multiple input files

Sometimes you have multiple input files that you want to merge for one sample. For example, a common use case is a single library that was spread across multiple sequencing lanes, yielding multiple input files that need to be merged, and then run through the pipeline as one. Rather than putting multiple lines in your sample annotation sheet, which causes conceptual and analytical challenges, PEP has two ways to merge these:

1. Use shell expansion characters (like '*' or '[]') in your *data_source* definition or filename (good for simple merges)
2. Specify a *merge table* which maps input files to samples for samples with more than one input file (infinitely customizable for more complicated merges).

Dealing with multiple input files is described in detail in the PEP documentation.

Note: to handle different *classes* of input files, like read1 and read2, these are *not* merged and should be handled as different derived columns in the main sample annotation sheet (and therefore different arguments to the pipeline).

1.11 Usage reference

Looper doesn't just run pipelines; it can also check and summarize the progress of your jobs, as well as remove all files created by them.

Each task is controlled by one of the five main commands `run`, `summarize`, `destroy`, `check`, `clean`.

- `looper run`: Runs pipelines for each sample, for each pipeline. This will use your `compute` settings to build and submit scripts to your specified compute environment, or run them sequentially on your local computer.
- `looper summarize`: Summarize your project results. This command parses all key-value results reported in the each sample *stats.tsv* and collates them into a large summary matrix, which it saves in the project output directory. This creates such a matrix for each pipeline type run on the project, and a combined master summary table.
- `looper check`: Checks the run progress of the current project. This will display a summary of job status; which pipelines are currently running on which samples, which have completed, which have failed, etc.
- `looper destroy`: Deletes all output results for this project.

Here you can see the command-line usage instructions for the main looper command and for each subcommand:

1.11.1 looper --help

```
version: 0.8.1
usage: looper [-h] [-V] [--logfile LOGFILE] [--verbosity {0,1,2,3,4}] [--dbg]
              {run,summarize,destroy,check,clean} ...

looper - Loop through samples and submit pipelines.

positional arguments:
  {run,summarize,destroy,check,clean}
    run                 Main Looper function: Submit jobs for samples.
    summarize           Summarize statistics of project samples.
    destroy             Remove all files of the project.
    check               Checks flag status of current runs.
    clean               Runs clean scripts to remove intermediate files of
                        already processed jobs.

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  --logfile LOGFILE     Optional output file for looper logs (default: None)
  --verbosity {0,1,2,3,4}
                        Choose level of verbosity (default: None)
  --dbg                 Turn on debug mode (default: False)

For subcommand-specific options, type: 'looper <subcommand> -h'
https://github.com/peppykit/looper
```

1.11.2 looper run --help

```
version: 0.8.1
usage: looper run [-h] [-t TIME_DELAY] [--ignore-flags] [--compute COMPUTE]
                  [--env ENV] [--limit LIMIT] [--lump LUMP] [--lumpn LUMPN]
                  [--file-checks] [-d]
                  [--exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                  | --include-protocols
                  [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]]
                  [--sp SUBPROJECT]
                  config_file

Main Looper function: Submit jobs for samples.

positional arguments:
  config_file           Project configuration file (YAML).

optional arguments:
  -h, --help            show this help message and exit
  -t TIME_DELAY, --time-delay TIME_DELAY
                        Time delay in seconds between job submissions.
  --ignore-flags        Ignore run status flags? Default: False. By default,
                        pipelines will not be submitted if a pypiper flag file
                        exists marking the run (e.g. as 'running' or
                        'failed'). Set this option to ignore flags and submit
                        the runs anyway.
  --compute COMPUTE     YAML file with looper environment compute settings.
  --env ENV             Employ looper environment compute settings.
```

```
  --limit LIMIT         Limit to n samples.
  --lump LUMP           Maximum total input file size for a lump/batch of
                        commands in a single job
  --lumpn LUMPN         Number of individual scripts grouped into single
                        submission
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                        Operate only on samples that either lack a protocol or
                        for which protocol is not in this collection.
  --include-protocols [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]
                        Operate only on samples associated with these
                        protocols; if not provided, all samples are used.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

1.11.3 looper summarize --help

```
version: 0.8.1
usage: looper summarize [-h] [--file-checks] [-d]
                        [--exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...
↪]]
                        | --include-protocols
                        [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]]
                        [--sp SUBPROJECT]
                        config_file

Summarize statistics of project samples.

positional arguments:
  config_file           Project configuration file (YAML).

optional arguments:
  -h, --help            show this help message and exit
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                        Operate only on samples that either lack a protocol or
                        for which protocol is not in this collection.
  --include-protocols [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]
                        Operate only on samples associated with these
                        protocols; if not provided, all samples are used.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

1.11.4 looper destroy --help

```
version: 0.8.1
usage: looper destroy [-h] [--file-checks] [-d]
                      [--exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                      | --include-protocols
                      [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]]
                      [--sp SUBPROJECT]
                      config_file
```

```
Remove all files of the project.

positional arguments:
  config_file           Project configuration file (YAML).

optional arguments:
  -h, --help            show this help message and exit
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                        Operate only on samples that either lack a protocol or
                        for which protocol is not in this collection.
  --include-protocols [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]
                        Operate only on samples associated with these
                        protocols; if not provided, all samples are used.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

1.11.5 looper check --help

```
version: 0.8.1
usage: looper check [-h] [-A] [-F [FLAGS [FLAGS ...]]] [--file-checks] [-d]
                    [--exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                    | --include-protocols
                    [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]]
                    [--sp SUBPROJECT]
                    config_file

Checks flag status of current runs.

positional arguments:
  config_file           Project configuration file (YAML).

optional arguments:
  -h, --help            show this help message and exit
  -A, --all-folders     Check status for all project's output folders, not
                        just those for samples specified in the config file
                        used
  -F [FLAGS [FLAGS ...]], --flags [FLAGS [FLAGS ...]]
                        Check on only these flags/status values.
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                        Operate only on samples that either lack a protocol or
                        for which protocol is not in this collection.
  --include-protocols [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]
                        Operate only on samples associated with these
                        protocols; if not provided, all samples are used.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

1.11.6 looper clean --help

```
version: 0.8.1
usage: looper clean [-h] [--file-checks] [-d]
                    [--exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                    | --include-protocols
                    [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]]
                    [--sp SUBPROJECT]
                    config_file

Runs clean scripts to remove intermediate files of already processed jobs.

positional arguments:
  config_file           Project configuration file (YAML).

optional arguments:
  -h, --help            show this help message and exit
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --exclude-protocols [EXCLUDE_PROTOCOLS [EXCLUDE_PROTOCOLS ...]]
                        Operate only on samples that either lack a protocol or
                        for which protocol is not in this collection.
  --include-protocols [INCLUDE_PROTOCOLS [INCLUDE_PROTOCOLS ...]]
                        Operate only on samples associated with these
                        protocols; if not provided, all samples are used.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

1.12 Configuration files

Looper uses YAML configuration files for several purposes. Looper is designed to be organized, modular, and very configurable, so there are several configuration files. We've organized the configuration files so they each handle a different level of infrastructure: environment, project, sample, or pipeline. This makes the system very adaptable and portable, but for a newcomer, it is easy to confuse what the different configuration files are used for. So, here's an explanation of each for you to use as a reference until you are familiar with the whole ecosystem. Which ones you need to know about will depend on whether you're a pipeline user (running pipelines on your project) or a pipeline developer (building your own pipeline).

1.12.1 Pipeline users

Users (non-developers) of pipelines only need to be aware of one or two YAML files:

- *project config file*: This file is specific to each project and contains information about the project's metadata, where the processed files should be saved, and other variables that allow to configure the pipelines specifically for this project. This file follows the standard looper format (now referred to as `PEP` format).

If you are planning to submit jobs to a cluster, then you need to know about a second YAML file:

- *PEPENV environment config*: This file tells looper how to use compute resource managers, like SLURM. This file doesn't require much editing or maintenance beyond initial setup.

That should be all you need to worry about as a pipeline user. If you need to adjust compute resources or want to develop a pipeline or have more advanced project-level control over pipelines, then you'll need to know about a few others:

1.12.2 Pipeline developers

If you want to add a new pipeline to looper, tweak the way looper interacts with a pipeline for a given project, or change the default cluster resources requested by a pipeline, then you need to know about a configuration file that coordinates linking your pipeline in to your looper project.

- *pipeline interface file*: Has two sections: 1) `protocol_mapping` tells looper which pipelines exist, and how to map each protocol (sample data type) to its pipelines; 2) `pipelines` links looper to the pipelines by describing options, arguments, and compute resources that the pipeline needs to run.

Finally, if you're using Pypiper to develop pipelines, it uses a pipeline-specific configuration file (detailed in the Pypiper documentation):

- Pypiper pipeline config file: Each pipeline may (optionally) provide a configuration file describing where software is, and parameters to use for tasks within the pipeline. This configuration file is by default named identical to the pypiper script name, with a *.yaml* extension instead of *.py* (So *rna_seq.py* looks for an accompanying *rna_seq.yaml* file by default). These files can be changed on a per-project level using the *pipeline_config* section in your project config file.

1.13 FAQ

- **Why isn't the `looper` executable in my path?** By default, Python packages are installed to `~/.local/bin`. You can add this location to your path by appending it (`export PATH=$PATH:~/.local/bin`).
- **How can I run my jobs on a cluster?** See *cluster resource managers*.
- **Which configuration file has which settings?** Here's a list: *config files*
- **What's the difference between *looper* and *pypiper*?** Pypiper and Looper work together as a comprehensive pipeline management system. Pypiper builds individual, single-sample pipelines that can be run one sample at a time. Looper then processes groups of samples, submitting appropriate pipelines to a cluster or server. The two projects are independent and can be used separately, but they are most powerful when combined.
- **Why isn't looper submitting my pipeline: `Not submitting, flag found: ['*_completed.flag']`?** When using `looper run`, looper first checks the sample output for flag files (which can be *_completed.flag*, or *_running.flag*, or *_failed.flag*). Typically, we don't want to resubmit a job that's already running or already finished, so by default, looper **will not submit a job when it finds a flag file**. This is what the message above is indicating. If you do in fact want to re-rerun a sample (maybe you've updated the pipeline, or you want to run restart a failed attempt), you can do so by just passing `--ignore-flags` to looper at startup. This will skip the flag check **for all samples**. If you only want to re-run or restart a few samples, it's best to just delete the flag files for the samples you want to restart, then use `looper run` as normal.
- **How can I resubmit a subset of jobs that failed?** By default, looper **will not submit a job that has already run**. If you want to re-rerun a sample (maybe you've updated the pipeline, or you want to run restart a failed attempt), you can do so by passing `--ignore-flags` to looper at startup, but this will **resubmit all samples**. If you only want to re-run or restart a few samples, it's best to just delete the flag files manually for the samples you want to restart, then use `looper run` as normal.
- **Can I pass additional command-line arguments to my pipeline on-the-fly?** Yes! Any command-line arguments passed to *looper run that are not consumed by looper* will simply be handed off untouched to *all the pipelines*. This gives you a handy way to pass-through command-line arguments that you want passed to every job in a given looper run. For example, you may run *looper run config.yaml -R* – since *looper 'does not understand '-R*, this will be passed to every pipeline.

For example, pypiper pipelines understand the *–recover* flag; so if you want to pass this flag through *looper* to all your pipeline runs, you may run *looper run config.yaml –recover*. Since *looper 'does not understand '–recover*, this will be passed to every pipeline. Obviously, this feature is limited to passing flags that *looper* does not understand, because those arguments will be consumed by *looper* and not passed through to individual pipelines.

- **How can I analyze my project interactively?** Looper uses the `peppy` package to model Project and Sample objects under the hood. These project objects are actually useful outside of looper. If you define your project using looper's *standardized project definition format* , you can use the project models to instantiate an in memory representation of your project and all of its samples, without using looper.

  If you're interested in this, you should check out the peppy package. All the documentation for model objects has moved there.

1.14 Changelog

- **v0.8.1** (*2018-04-02*):
  - Changed
    - Minor documentation and packaging updates for first Pypi release.
    - Fix a bug that incorrectly mapped protocols due to case sensitive issues
    - Fix a bug with `report_figure` that made it output pandas code
- **v0.8.0** (*2018-01-19*):
  - Changed
    - Use independent *peppy* package, replacing `models` module for core data types.
    - Integrate `ProtocolInterface` functionality into `PipelineInterface`.
- **v0.7.2** (*2017-11-16*):
  - Fixed
    - Correctly count successful command submissions when not using *–dry-run*.
- **v0.7.1** (*2017-11-15*):
  - Fixed
    - No longer falsely display that there's a submission failure.
    - Allow non-string values to be unquoted in the `pipeline_args` section.
- **v0.7** (*2017-11-15*):
  - New
    - Add `--lump` and `--lumpn` options
    - Catch submission errors from cluster resource managers
    - Implied columns can now be derived
    - Now protocols can be specified on the command-line *–include-protocols*
    - Add rudimentary figure summaries
    - Simplifies command-line help display
    - Allow wildcard protocol_mapping for catch-all pipeline assignment

        * Improve user messages
        * New sample_subtypes section in pipeline_interface
    - Changed
        * Sample child classes are now defined explicitly in the pipeline interface. Previously, they were guessed based on presence of a class extending Sample in a pipeline script.
        * Changed 'library' key sample attribute to 'protocol'

- **v0.6** (*2017-07-21*):
    - New
        * Add support for implied_column section of the project config file
        * Add support for Python 3
        * Merges pipeline interface and protocol mappings. This means we now allow direct pointers to `pipeline_interface.yaml` files, increasing flexibility, so this relaxes the specified folder structure that was previously used for `pipelines_dir` (with `config` subfolder).
        * Allow URLs as paths to sample sheets.
        * Allow tsv format for sample sheets.
        * Checks that the path to a pipeline actually exists before writing the submission script.
    - Changed
        * Changed LOOPERENV environment variable to PEPENV, generalizing it to generic models
        * Changed name of `pipelines_dir` to `pipeline_interfaces` (but maintained backwards compatibility for now).
        * Changed name of `run` column to `toggle`, since `run` can also refer to a sequencing run.
        * Relaxes many constraints (like resources sections, pipelines_dir columns), making project configuration files useful outside looper. This moves us closer to dividing models from looper, and improves flexibility.
        * Various small bug fixes and dev improvements.
        * Require *setuptools* for installation, and *pandas 0.20.2*. If *numexpr* is installed, version *2.6.2* is required.
        * Allows tilde in `pipeline_interfaces`

- **v0.5** (*2017-03-01*):
    - New
        * Add new looper version tracking, with *–version* and *-V* options and printing version at runtime
        * Add support for asterisks in file paths
        * Add support for multiple pipeline directories in priority order
        * Revamp of messages make more intuitive output
        * Colorize output
        * Complete rehaul of logging and test infrastructure, using logging and pytest packages
    - Changed
        * Removes pipelines_dir requirement for models, making it useful outside looper

        * Small bug fixes related to *all_input_files* and *required_input_files* attributes
        * More robust installation and more explicit requirement of Python 2.7

- **v0.4** (*2017-01-12*):
    - New
        * New command-line interface (CLI) based on sub-commands
        * New subcommand (`looper summarize`) replacing the `summarizePipelineStats.R` script
        * New subcommand (`looper check`) replacing the `flagCheck.sh` script
        * New command (`looper destroy`) to remove all output of a project
        * New command (`looper clean`) to remove intermediate files of a project flagged for deletion
        * Support for portable and pipeline-independent allocation of computing resources with Looperenv.
    - Changed
        * Removed requirement to have `pipelines` repository installed in order to extend base Sample objects
        * Maintenance of sample attributes as provided by user by means of reading them in as strings (to be improved further)
        * Improved serialization of Sample objects

1.15 Support

Please use the issue tracker at GitHub to file bug reports or feature requests: https://github.com/epigen/looper/issues.

1.16 Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Links

- Public-facing permalink: http://databio.org/looper
- Documentation: http://looper.readthedocs.io/
- Source code: http://github.com/pepkit/looper