pipelines Documentation

Release 0.6.0

Bock lab

Aug 09, 2017

Getting Started

Deploying pipelines just got easier. Looper is a python application that deploys pipelines across samples with minimal effort. Looper is **not** a pipeline development framework; it does not help develop pipelines, but sits a layer above the pipeline to manage projects and samples for any type of pipeline. 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

Introduction

Looper is a job submitting engine. Do not confuse it with a pipeline workflow engine, which is used to build pipelines. Looper assumes you already have pipelines built, and it helps you map samples to those pipelines. If you have a pipeline and a bunch of samples you want to run, looper can help you organize the inputs and outputs.

It's scalable: by default, it runs your jobs sequentially on the local computer, but with a small configuration change, it will create and submit jobs to any cluster resource manager (like SLURM, SGE, or LFS).

The basics: 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. Easy.

Looper is modular and totally configurable, so it scales as your needs grow. We provide sensible defaults for ease-of-use, but you can configure just about anything. You have complete control. `Looper` handles the mundane project organization tasks that you don't want to worry about.

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/epigen/looper/zipball/master
```

Update looper with:

```
pip install --user --upgrade https://github.com/epigen/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/epigen/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` – check out the *FAQ*.

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

Features

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 defines a flexible standard format for describing projects, and there are other tools that can read these same formats. For example, we are working on an R package that will read the same project definition and provide all your sample metadata (and pipeline results) in an R analysis environment, with no additional effort. 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 lets you easily use samples with input files on different file systems or from different projects, with different naming conventions. This also makes it easy to share projects across compute environments without having to change sample annotations.
- **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.

Usage

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:

looper --help

```
version: 0.6.0
usage: looper [-h] [-V] {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

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

looper run --help

```
version: 0.6.0
usage: looper run [-h] [-t TIME_DELAY] [--ignore-flags] [--compute COMPUTE]
                  [--env ENV] [--limit LIMIT] [--file-checks] [-d]
                  [--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.
  --file-checks         Perform input file checks. Default=True.
  -d, --dry-run         Don't actually submit the project/subproject.
  --sp SUBPROJECT       Name of subproject to use, as designated in the
                        project's configuration file
```

looper summarize --help

```
version: 0.6.0
usage: looper summarize [-h] [--file-checks] [-d] [--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.
  --sp SUBPROJECT  Name of subproject to use, as designated in the project's
                   configuration file
```

looper destroy --help

```
version: 0.6.0
usage: looper destroy [-h] [--file-checks] [-d] [--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.
  --sp SUBPROJECT  Name of subproject to use, as designated in the project's
                   configuration file
```

looper check --help

```
version: 0.6.0
usage: looper check [-h] [--file-checks] [-d] [--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
  --file-checks    Perform input file checks. Default=True.
  -d, --dry-run    Don't actually submit the project/subproject.
  --sp SUBPROJECT  Name of subproject to use, as designated in the project's
                   configuration file
```

looper clean --help

```
version: 0.6.0
usage: looper clean [-h] [--file-checks] [-d] [--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.
  --sp SUBPROJECT  Name of subproject to use, as designated in the project's
                   configuration file
```

looper --help --details

```
version: 0.6.0
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/epigen/looper
```

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

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` - a directory containing one directory with the output of the pipelines, for each sample.
- `submissions` - which holds yaml representations of the samples and log files of the submited 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!

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 pipeline to looper*.

How to define a project

To use `looper` with your project, you must define your project using Looper's standard project definition format. If you follow this format, then your project can be read not only by looper for submitting pipelines, but also for other tasks, like: summarizing pipeline output, analysis in R (using the `project.init` package), or building UCSC track hubs.

The format is simple and modular, so you only need to define the components you plan to use. You need to supply 2 files:

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

Quick example: In the simplest case, `project_config.yaml` is just a few lines of `yaml`. Here's a minimal example **project_config.yaml**:

```
metadata:
  sample_annotation: /path/to/sample_annotation.csv
  output_dir: /path/to/output/folder
  pipeline_interfaces: path/to/pipeline_interface.yaml
```

The **output_dir** key specifies where to save results. The **pipeline_interfaces** key points to your looper-compatible pipelines (described in *linking the pipeline interface*). The **sample_annotation** key points to another file, which is a comma-separated value (`csv`) file describing samples in the project. Here's a small example of **sample_annotation.csv**:

Table 1.1: Minimal Sample Annotation Sheet

sample_name	library	file
frog_1	RNA-seq	frog1.fq.gz
frog_2	RNA-seq	frog2.fq.gz
frog_3	RNA-seq	frog3.fq.gz
frog_4	RNA-seq	frog4.fq.gz

With those two simple files, you could run looper, and that's fine for just running a quick test on a few files. In practice, you'll probably want to use some of the more advanced features of looper by adding additional information to your configuration `yaml` file and your sample annotation `csv` file.

For example, by default, your jobs will run serially on your local computer, where you're running `looper`. If you want to submit to a cluster resource manager (like SLURM or SGE), you just need to specify a `compute` section.

Let's go through the more advanced details of both annotation sheets and project config files:

orphan

Sample annotation sheet

The `sample annotation sheet` is a csv file containing information about all samples in a project. This should be regarded as an immutable and the most important piece of metadata in a project. **One row corresponds to one sample** (or, more specifically, one pipeline run).

A sample annotation sheet may contain any number of columns you need for your project. You can think of these columns as *sample attributes*, and you may use these columns later in your pipelines or analysis (for example, you could define a column called `organism` and use this to adjust the reference genome to use for each sample).

Special columns

Certain keyword columns are required or provide looper-specific features. Any additional columns become attributes of your sample and will be part of the project's metadata for the samples. Mostly, you have complete control over any other column names you want to add, but there are a few reserved column names:

- `sample_name` - a **unique** string identifying each sample[1]. This is **required** for `Sample` construction. The only required column.
- `organism` - a string identifying the organism ("human", "mouse", "mixed"). **Recommended** but not required.
- `library` - While not needed to build a `Sample`, this column is required for submission of job(s). It specifies the source of data for the sample (*e.g.* ATAC-seq, RNA-seq, RRBS). `Looper` uses this information to determine which pipelines are relevant for the `Sample`.
- `data_source` - This column is used by default to specify the location of the input data file. Usually you want your annotation sheet to specify the locations of files corresponding to each sample. You can use this to simplify pointing to file locations with a neat string-replacement method that keeps things clean and portable. For more details, see the advanced section *Derived columns*. Really, you just need any column specifying at least 1 data file for input. This is **required** for `Looper` to submit job(s) for a `Sample`.
- `toggle` - If the value of this column is not 1, Looper will not submit the pipeline for that sample. This enables you to submit a subset of samples.

Here are a few example annotation sheets:

Table 1.2: Example Sample Annotation Sheet

sample_name	library	organism	ip	data_source
atac-seq_PE	ATAC-seq	human		microtest
atac-seq_SE	ATAC-seq	human		microtest
chip-seq_PE	CHIP-seq	human	H3K27ac	microtest
chip-seq_SE	CHIP-seq	human	H3K27ac	microtest
chipmentation_PE	ChIPmentation	human	H3K27ac	microtest
chipmentation_SE	ChIPmentation	human	H3K27ac	microtest
cpgseq_example_data	CpG-seq	human		microtest
quant-seq_SE	Quant-seq	human		microtest
rrbs	RRBS	human		microtest
rrbs_PE	RRBS	human		microtest
wgbs	WGBS	human		microtest
RNA_TRUseq_50SE	SMART	human		microtest
RNA_SMART_50SE	SMART	human		microtest
rrbs_PE_fq	RRBS	human		microtest
rrbs_fq	RRBS	human		microtest

[1] This should be a string without whitespace (space, tabs, etc...). If it contains whitespace, an error will be thrown. Similarly, looper will not allow any duplicate entries under sample_name.

Table 1.3: Example Sample Annotation Sheet

sample_name	library	organism	flowcell	lane	BSF_name	data_source
albt_0h	RRBS	albatross	BSFX0190	1	albt_0h	bsf_sample
albt_1h	RRBS	albatross	BSFX0190	1	albt_1h	bsf_sample
albt_2h	RRBS	albatross	BSFX0190	1	albt_2h	bsf_sample
albt_3h	RRBS	albatross	BSFX0190	1	albt_3h	bsf_sample
frog_0h	RRBS	frog				frog_data
frog_1h	RRBS	frog				frog_data
frog_2h	RRBS	frog				frog_data
frog_3h	RRBS	frog				frog_data

orphan

Project config file

A minimal project config file requires very little; only a single section (`metadata` – see above). Here are additional details on this and other optional project config file sections:

Project config section: metadata

The *metadata* section contains paths to various parts of the project: the output directory (the parent directory), the results subdirector, the submission subdirectory (where submit scripts are stored), and pipeline scripts.Pointers to sample annotation sheets. This is the only required section.

Example:

```
metadata:
  sample_annotation: /path/to/sample_annotation.csv
  output_dir: /path/to/output/folder
  pipeline_interfaces: /path/to/pipeline_interface.yaml
```

Project config section: data_sources

The *data_sources* section uses regex-like commands to point to different spots on the filesystem for data. The variables (specified by `{variable}`) are populated by sample attributes (columns in the sample annotation sheet). You can also use shell environment variables (like `${HOME}`) in these.

Example:

```
data_sources:
  source1: /path/to/raw/data/{sample_name}_{sample_type}.bam
  source2: /path/from/collaborator/weirdNamingScheme_{external_id}.fastq
  source3: ${HOME}/{test_id}.fastq
```

For more details, see *Derived columns*.

Project config section: derived_columns

`derived_columns` is just a simple list that tells looper which column names it should populate as data_sources. Corresponding sample attributes will then have as their value not the entry in the table, but the value derived from the

string replacement of sample attributes specified in the config file. This enables you to point to more than one input file for each sample (for example read1 and read2).

Example:

```
derived_columns: [read1, read2, data_1]
```

For more details, see *Derived columns*.

Project config section: implied_columns

`implied_columns` lets you infer additional attributes, which can be useful for pipeline arguments.

Example:

```
implied_columns:
  organism:
    human:
      genome: "hg38"
      macs_genome_size: "hs"
```

For more details, see *Implied columns*.

Project config section: subprojects

Subprojects are useful to define multiple similar projects within a single project config file. Under the subprojects key, you can specify names of subprojects, and then underneath these you can specify any project config variables that you want to overwrite for that particular subproject. Tell looper to load a particular subproject by passing `--sp subproject-name` on the command line.

For example:

```
subprojects:
  diverse:
        metadata:
          sample_annotation: psa_rrbs_diverse.csv
  cancer:
        metadata:
          sample_annotation: psa_rrbs_intracancer.csv
```

This project would specify 2 subprojects that have almost the exact same settings, but change only their `metadata.sample_annotation` parameter (so, each subproject points to a different sample annotation sheet). Rather than defining two 99% identical project config files, you can use a subproject.

Project config section: pipeline_config

Occasionally, a particular project needs to run a particular flavor of a pipeline. Rather than creating an entirely new pipeline, you can parameterize the differences with a **pipeline config** file, and then specify that file in the **project config** file.

Example:

```
pipeline_config:
  # pipeline configuration files used in project.
  # Key string must match the _name of the pipeline script_ (including extension)
  # Relative paths are relative to this project config file.
```

```
  # Default (null) means use the generic config for the pipeline.
  rrbs.py: null
  # Or you can point to a specific config to be used in this project:
  wgbs.py: wgbs_flavor1.yaml
```

This will instruct *looper* to pass *-C wgbs_flavor1.yaml* to any invocations of wgbs.py (for this project only). Your pipelines will need to understand the config file (which will happen automatically if you use pypiper).

Project config section: pipeline_args

Sometimes a project requires tweaking a pipeline, but does not justify a completely separate **pipeline config** file. For simpler cases, you can use the *pipeline_args* section, which lets you specify command-line parameters via the project config. This lets you fine-tune your pipeline, so it can run slightly differently for different projects.

Example:

```
pipeline_args:
  rrbs.py:  # pipeline identifier: must match the name of the pipeline script
        # here, include all project-specific args for this pipeline
        "--flavor": simple
        "--flag": null
```

For flag-like options (options without parameters), you should set the value to the yaml keyword `null` (which means no value). Looper will pass the key to the pipeline without a value. The above specification will now pass `--flavor=simple` and `--flag` (no parameter) whenever `rrbs.py` is invoked – for this project only. This is a way to control (and record!) project-level pipeline arg tuning. The only keyword here is *pipeline_args*; all other variables in this section are specific to particular pipelines, command-line arguments, and argument values.

Project config section: compute

You can specify project-specific compute settings in a `compute` section. However, you're better off specifying this globally using a `pepenv` environment configuration. Instructions are at the pepenv repository. If you do need project-specific control over compute settings (like submitting a certain project to a certain resource account), you can do this by specifying variables in a project config `compute` section, which will override global pepenv values for that project only.

```
compute:
  partition: project_queue_name
```

Project config section: track_configurations

Warning: The *track_configurations* section is for making UCSC trackhubs. This is a work in progress that is functional, but ill-documented, so it is best avoided for now.

Project config complete example

Here's an example. Additional fields can be added as well and will be ignored.

```
    metadata:
      # Relative paths are considered relative to this project config file.
      # Typically, this project config file is stored with the project metadata
```

```
        # sample_annotation: one-row-per-sample metadata
        sample_annotation: table_experiments.csv
        # merge_table: input for samples with more than one input file
        merge_table: table_merge.csv
        # compare_table: comparison pairs or groups, like normalization samples
        compare_table: table_compare.csv
        # output_dir: the parent, shared space for this project where results go
        output_dir: /fhgfs/groups/lab_bock/shared/projects/example
        # results and submission subdirs are subdirectories under parent output_dir
        # results: where output sample folders will go
        # submission: where cluster submit scripts and log files will go
        results_subdir: results_pipeline
        submission_subdir: submission
        # pipeline_interfaces: the pipeline_interface.yaml file or files for Looper␣
↪pipelines
        # scripts (and accompanying pipeline config files) for submission.
        pipeline_interfaces: /path/to/shared/projects/example/pipeline_interface.yaml


      data_sources:
        # specify the ABSOLUTE PATH of input files using variable path expressions
        # entries correspond to values in the data_source column in sample_annotation␣
↪table
        # {variable} can be used to replace environment variables or other sample_
↪annotation columns
        # If you use {variable} codes, you should quote the field so python can parse␣
↪it.
        bsf_samples: "$RAWDATA/{flowcell}/{flowcell}_{lane}_samples/{flowcell}_{lane}#
↪{BSF_name}.bam"
        encode_rrbs: "/path/to/shared/data/encode_rrbs_data_hg19/fastq/{sample_name}.
↪fastq.gz"


implied_columns:
      # supported genomes/transcriptomes and organism -> reference mapping
  organism:
    human:
      genome: hg38
      transcriptome: hg38_cdna
    mouse:
      genome: mm10
      transcriptome: mm10_cdna

      pipeline_config:
        # pipeline configuration files used in project.
        # Default (null) means use the generic config for the pipeline.
        rrbs: null
        # Or you can point to a specific config to be used in this project:
        # rrbs: rrbs_config.yaml
        # wgbs: wgbs_config.yaml
        # cgps: cpgs_config.yaml
```

Derived columns

On your sample sheet, you will need to point to the input file or files for each sample. Of course, you could just add a column with the file path, like `/path/to/input/file.fastq.gz`. For example:

Table 1.4: Sample Annotation Sheet (bad example)

sample_name	library	organism	time	file_path
pig_0h	RRBS	pig	0	/data/lab/project/pig_0h.fastq
pig_1h	RRBS	pig	1	/data/lab/project/pig_1h.fastq
frog_0h	RRBS	frog	0	/data/lab/project/frog_0h.fastq
frog_1h	RRBS	frog	1	/data/lab/project/frog_1h.fastq

This is common, and it works in a pinch with Looper, but what if the data get moved, or your filesystem changes, or you switch servers or move institutes? Will this data still be there in 2 years? Do you want long file paths cluttering your annotation sheet? What if you have 2 or 3 input files? Do you want to manually manage these unwieldy absolute paths?

`Looper` makes it really easy to do better: you can make one or your annotation columns into a flexible `derived column` that will be populated based on a source template you specify in the `project_config.yaml`. What was originally `/long/path/to/sample.fastq.gz` would instead contain just a key, like `source1`. Columns that use a key like this are called `derived columns`. Here's an example of the same sheet using a `derived column` for `file_path`:

Table 1.5: Sample Annotation Sheet (good example)

sample_name	library	organism	time	file_path
pig_0h	RRBS	pig	0	source1
pig_1h	RRBS	pig	1	source1
frog_0h	RRBS	frog	0	source1
frog_1h	RRBS	frog	1	source1

To do this, your project config file must specify two things: First, which columns are to be derived (in this case, `file_path`); and second, a `data_sources` section mapping keys to strings that will construct your path, like this:

```
derived_columns: [file_path]
data_sources:
  source1: /data/lab/project/{sample_name}.fastq
  source2: /path/from/collaborator/weirdNamingScheme_{external_id}.fastq
```

That's it! The source string can use other sample attributes (columns) using braces, as in `{sample_name}`. The attributes will be automatically populated separately for each sample. To take this a step further, you'd get the same result with this config file, which substitutes `{sample_name}` for other sample attributes, `{organism}` and `{time}`:

```
derived_columns: [file_path]
data_sources:
  source1: /data/lab/project/{organism}_{time}h.fastq
  source2: /path/from/collaborator/weirdNamingScheme_{external_id}.fastq
```

As long as your file naming system is systematic, you can easily deal with any external naming scheme, no problem at all. The idea is: don't put absolute paths to files in your annotation sheet. Instead, specify a data source and then provide a regex in the config file. This way if your data changes locations (which happens more often than we would like), or you change servers, or you want to share or publish the project, you just have to change the config file and not update paths in the annotation sheet; this makes the annotation sheet universal across environments, users, publication, etc. The whole project is now portable.

You can specify as many derived columns as you want (`data_source` is considered a derived column by default). An expression including any sample attributes (using `{attribute}`) will be populated for each of those columns.

Think of each sample as belonging to a certain type (for simple experiments, the type will be the same); then define the location of these samples in the project configuration file. As a side bonus, you can easily include samples from different locations, and you can also share the same sample annotation sheet on different environments (i.e. servers or users) by having multiple project config files (or, better yet, by defining a subproject for each environment). The only thing you have to change is the project-level expression describing the location, not any sample attributes (plus, you get to eliminate those annoying long/path/arguments/in/your/sample/annotation/sheet).

Check out the complete working example in the microtest repository.

Implied columns

At some point, you may have a situation where you need a single sample attribute (or column) to populate several different pipeline arguments with different values. In other words, the value of a given attribute may **imply** values for other attributes. It would be nice if you didn't have to enumerate all of these secondary, implied attributes, and could instead just infer them from the value of the original attribute. For example, if my *organism* attribute is `human`, this implies a few other secondary attributes (which may be project-specific): For one project, I want to set `genome` to `hg38` and `macs_genome_size` to `hs`. Of course, I could just define columns called `genome` and `macs_genome_size`, but these would be invariant, so it feels inefficient and unweildy; and then, changing the aligned genome would require changing the sample annotation sheet (every sample, in fact). You can do this with looper, of course, but a better way would be handle these things at the project level.

As a more elegant alternative, Looper offers a `project_config` section called `implied_columns`. Instead of hard-coding `genome` and `macs_genome_size` in the sample annotation sheet, you can simply specify that the attribute `organism` **implies** additional attribute-value pairs (which may vary by sample based on the value of the `organism` attribute). This lets you specify the genome, transcriptome, genome size, and other similar variables all in your project configuration file.

To do this, just add an `implied_columns` section to your project_config.yaml file. Example:

```
implied_columns:
  organism:
    human:
      genome: "hg38"
      macs_genome_size: "hs"
    mouse:
      genome: "mm10"
      macs_genome_size: "mm"
```

There are 3 layers in the `implied_columns` hierarchy. The first layer, (sub-values under `implied_columns`; here, `organism`), are primary columns from which new attributes will rely. The second layer (here, `human` or `mouse`) are possible values your samples may take in the primary column. The third layer (`genome` and `macs_genome_size`) are the key-value pair of new, implied columns for any samples with the required value for that primary column. In this example, any samples with organism set to "human" will automatically also have attributes for genome (hg38) and for macs_genome_size (hs). Any samples with organism set to "mouse" will have the corresponding values. A sample with organism set to `frog` would lack attributes for `genome` and `macs_genome_size`, since those columns are not implied by `frog`.

This system essentially lets you set global, species-level attributes at the project level instead of duplicating that information for every sample that belongs to a species. Even better, it's generic – so you can do this for any subdivision of samples (just replace `organism` with whatever you like). This makes your project more portable and does a better job conceptually with separating sample attributes from project attributes; after all, a reference genome assembly is really not an inherent property of a sample, but of a sample in respect to a particular project or alignment.

Cluster computing

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.

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).

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*.

Advanced features

Handling 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, we introduce 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).

To do the first option, just change your data source specifications, like this:

```
data_R1: "${DATA}/{id}_S{nexseq_num}_L00*_R1_001.fastq.gz"
data_R2: "${DATA}/{id}_S{nexseq_num}_L00*_R2_001.fastq.gz"
```

To do the second option, just provide a merge table in the *metadata* section of your project config:

metadata: merge_table: mergetable.csv

Make sure the `sample_name` column of this table matches, and then include any columns you need to point to the data. `Looper` will automatically include all of these files as input passed to the pipelines. Warning: do not use both of these options simultaneously for the same sample, it will lead to multiple merges.

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).

Connecting to multiple pipelines

If you have a project that contains samples of different types, then you may need to specify multiple pipeline repositories to your project. Starting in version 0.5, looper can handle a priority list of pipelines. Starting with version 0.6, these pointers should point directly at a pipeline interface files (instead of at directories as previously). in the metadata.pipeline_interfaces attribute.

For example:

```
metadata:
  pipeline_interfaces: [pipeline_iface1.yaml, pipeline_iface2.yaml]
```

In this case, for a given sample, looper will first look in pipeline_iface1.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.yaml file). Having submitted a suitable pipeline it will ignore the pipeline_iface2.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.

How to link a pipeline to your project

Looper links to pipelines through a file called the *pipeline_interface*. If you're using pre-made looper pipelines, you don't need to create a new interface; you just use the one that comes with the pipeline. If you need to link a new pipeline to looper, then you'll need to create a new pipeline interface file. The instructions below show you how to use pipelines in either category.

Linking a looper-compatible pipeline

Many projects will require only existing pipelines that are already looper-compatible. We maintain a (growing) list of known publicly available looper-compatible pipelines that will give you a good place to start. This list includes pipelines for data types like RNA-seq, bisulfite sequencing, etc.

To use one of these pipelines, just clone the repository and the point your project to that pipeline's *pipeline_interface* file. You do this with 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. After that, you just need to make sure your project definition provides all the necessary sample metadata that is required by the pipeline you want to use. 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.

Linking a custom pipeline

Hint: If you're just a pipeline **user**, you don't need to worry about this section. This is for those who develop pipelines, or those who want to use a currently defined looper project to submit to an existing pipeline that isn't already configured for looper.

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:

orphan

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
```

- **Case 3:** a protocol runs one pipeline which depends on another. Example: `WGBSNM: first;second; third;(fourth, fifth)`

Warning: This feature (pipeline dependency) is not implemented yet. This documentation describes a protocol that may be implemented in the future, if it is necessary to have dependency among pipeline submissions.

The basic format for pipelines run simultaneously is: `PROTOCOL: pipeline1 [, pipeline2, ...]`. Use semi-colons to indicate dependency.

Example `protocol_mapping` section:

```
WGBSQC: >
  wgbs.py;
  (nnm.py, pdr.py)
```

orphan

Pipeline interface section: pipelines

The `pipelines` section specifies command-line arguments required by the pipeline. In addition, if you're using a cluster resource manager, it also specifies which compute resources to request. For each pipeline, you specify variables (some optional and some required). The possible attributes to specify for each pipeline include:

- `name` (recommended): Name of the pipeline. This is used to assess pipeline flags (if your pipeline employs them, like pypiper pipelines).
- `arguments` (required): List of key-value pairs of arguments, and attribute sources to pass to the pipeline. The key corresponds verbatim to the string that will be passed on the command line to the pipeline. 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). For flag-like arguments that lack a value, you may specify *null* as the value (e.g. *"–quiet-mode": null*).
- `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 environment variables (like `${HOME}`) in the `path`.
- `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.

Example:

```
pipeline_script.py:  # this is variable (script filename)
  name: value  # used for assessing pipeline flags (optional)
  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"
```

Full example:

```
rrbs.py:
  name: RRBS
  looper_args: True
  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"
```

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).

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:

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*.

Project models

Looper uses object oriented programming (OOP) under the hood. This means that concepts like a sample to be processed or a project are modeled as objects in Python. These project objects are actually useful outside of looper. At some point, we will likely separate the project objects into their own Python package, but for now, you can use them independently of looper, even though they are embedded within the looper package. They are functionally independent.

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. Here is a brief description of how you would do this.

```
from looper import models

my_project = models.Project("path/to/project_config.yaml")
my_samples = my_project.samples
```

Once you have your project and samples in your Python session, the possibilities are endless. This is the way looper reads your project; looper uses these objects to loop through each sample and submit pipelines for each. You could just as easily use these objects for other purposes; for example, one way we use these objects is for post-pipeline processing. After we use looper to run each sample through its pipeline, we can load the project and it sample objects into an analysis session, where we do comparisons across samples. We are also working on on R package that will similarly read this standardized project definition format, giving you access to the same information within R.

This is a work in progress, but you can find more information and examples in the API.

Exploration:

To interact with the various `models` and become acquainted with their features and behavior, there is a lightweight module that provides small working versions of a couple of the core objects. Specifically, from within the `tests` directory, the Python code in the `tests.interactive` module can be copied and pasted into an interpreter. This provides a `Project` instance called `proj` and a `PipelineInterface` instance called `pi`. Additionally, this provides logging information in great detail, affording visibility into some what's happening as the `models` are created and used.

Extending sample objects

By default we use generic models (see the API for more) to handle samples in Looper, but these can also be reused in other contexts by importing `models` or by means of object serialization through YAML files.

Since these models provide useful methods to store, update, and read attributes in the objects created from them (most notably a *sample* - `Sample` object), a useful use case is during the run of a pipeline: a pipeline can create a more tailored `Sample` model, adding attributes or providing altered or additional methods.

Example:

You have several samples, of different experiment types, each yielding different varieties of data and files. For each sample of a given experiment type that uses a particular pipeline, the set of file path types that are relevant for the initial pipeline processing or for downstream analysis is known. For instance, a peak file with a certain genomic location will likely be relevant for a ChIP-seq sample, while a transcript abundance/quantification file will probably be used when working with a RNA-seq sample. This common situation, in which one or more file types are specific to a pipeline and analysis both benefits from and is amenable to a bespoke `Sample` *type*. Rather than working with a base

`Sample` instance and repeatedly specifying paths to relevant files, those locations can be provided just once, stored in an instance of the custom `Sample` *type*, and later used or modified as needed by referencing a named attribute on the object. This approach can dramatically reduce the number of times that a full filepath must be accurately keyed and thus saves some typing time. More significant, it's likely to save time lost to diagnostics of typo-induced errors. The most rewarding aspect of employing the `Sample` extension strategy, though, is a drastic readability boost. As the visual clutter of raw filepaths clears, code readers can more clearly focus on questions of *what* a filepath points to and *how* it's being used, rather than on the path itself.

Logistics:

It's the specification of *both an experiment or data type* ("library" or "protocol") *and a pipeline with which to process that input type* that `Looper` uses to determine which type of `Sample` object(s) to create for pipeline processing and analysis (i.e., which `Sample` extension to use). There's a pair of symmetric reasons for this–the relationship between input type and pipeline can be one-to-many, in both directions. That is, it's possible for a single pipeline to process more than one input type, and a single input type may be processed by more than one pipeline.

There are a few different `Sample` extension scenarios. Most basic is the one in which an extension, or *subtype*, is neither defined nor needed–the pipeline author does not provide one, and users do not request one. Almost equally effortless on the user side is the case in which a pipeline author intends for a single subtype to be used with her pipeline. In this situation, the pipeline author simply implements the subtype within the pipeline module, and nothing further is required–of the pipeline author or of a user! The `Sample` subtype will be found within the pipeline module, and the inference will be made that it's intended to be used as the fundamental representation of a sample within that pipeline. If a pipeline author extends the base `Sample` type in the pipeline module, it's likely that the pipeline's proper functionality depends on the use of that subtype. In a rare case, though, it may be desirable to use the base `Sample` type even if the pipeline author has provided a more customized version with her pipeline. To favor the base `Sample` over the tailored one created by a pipeline author, the user may simply set `sample_subtypes` to `null` in his own version of the pipeline interface, either for all types of inpute to that pipeline, or for just a subset of them. Read on for further information.

```
# atacseq.py

from models import Sample

class ATACseqSample(Sample):
        """
        Class to model ATAC-seq samples based on the generic Sample class.

        :param series: Pandas `Series` object.
        :type series: pandas.Series
        """

        def __init__(self, series):
                if not isinstance(series, pd.Series):
                        raise TypeError("Provided object is not a pandas Series.")
                super(ATACseqSample, self).__init__(series)
                self.make_sample_dirs()

        def set_file_paths(self):
                """Sets the paths of all files for this sample."""
                # Inherit paths from Sample by running Sample's set_file_paths()
                super(ATACseqSample, self).set_file_paths()

                self.fastqc = os.path.join(self.paths.sample_root, self.name + ".
↪fastqc.zip")
                self.trimlog = os.path.join(self.paths.sample_root, self.name + ".
↪trimlog.txt")
                self.fastq = os.path.join(self.paths.sample_root, self.name + ".fastq
↪")
```

```
                self.trimmed = os.path.join(self.paths.sample_root, self.name + ".
↪trimmed.fastq")
                self.mapped = os.path.join(self.paths.sample_root, self.name + ".
↪bowtie2.bam")
                self.peaks = os.path.join(self.paths.sample_root, self.name + "_peaks.
↪bed")
```

To leverage the power of a `Sample` subtype, the relevant model is the `PipelineInterface`. For each pipeline defined in the `pipelines` section of `pipeline_interface.yaml`, there's accommodation for a `sample_subtypes` subsection to communicate this information. The value for each such key may be either a single string or a collection of key-value pairs. If it's a single string, the value is the name of the class that's to be used as the template for each `Sample` object created for processing by that pipeline. If instead it's a collection of key-value pairs, the keys should be names of input data types (as in the `protocol_mapping`), and each value is the name of the class that should be used for each sample object of the corresponding key*for that pipeline*. This underscores that it's the *combination of a pipeline and input type* that determines the subtype.

```
# Content of pipeline_interface.yaml

protocol_mapping:
    ATAC: atacseq.py

pipelines:
    atacseq.py:
        ...
        ...
        sample_subtypes: ATACseqSample
        ...
        ...
    ...
    ...
```

If a pipeline author provides more than one subtype, the `sample_subtypes` section is needed to select from among them once it's time to create `Sample` objects. If multiple options are available, and the `sample_subtypes` section fails to clarify the decision, the base/generic type will be used. The responsibility for supplying the `sample_subtypes` section, as is true for the rest of the pipeline interface, therefore rests primarily with the pipeline developer. It is possible for an end user to modify these settings, though.

Since the mechanism for subtype detection is `inspect`-ion of each of the pipeline module's classes and retention of those which satisfy a subclass status check against `Sample`, it's possible for pipeline authors to implement a class hierarchy with multi-hop inheritance relationships. For example, consider the addition of the following class to the previous example of a pipeline module `atacseq.py`:

```
class DNaseSample(ATACseqSample):
    ...
```

In this case there are now two `Sample` subtypes available, and more generally, there will necessarily be multiple subtypes available in any pipeline module that uses a subtype scheme with multiple, serial inheritance steps. In such cases, the pipeline interface should include an unambiguous `sample_subtypes` section.

```
# Content of pipeline_interface.yaml

protocol_mapping:
    ATAC: atacseq.py
    DNase: atacseq.py

pipelines:
    atacseq.py:
```

```
        ...
        ...
        sample_subtypes:
            ATAC: ATACseqSample
            DNase: DNaseSample
        ...
        ...
    ...
    ...
```

API

looper.models

Project Models

Workflow explained:

- **Create a Project object**
  - Samples are created and added to project (automatically)

In the process, Models will check:

- Project structure (created if not existing)
- Existence of csv sample sheet with minimal fields
- Constructing a path to a sample's input file and checking for its existence
- Read type/length of samples (optionally)

Example:

```
from models import Project
prj = Project("config.yaml")
# that's it!
```

Explore:

```
# see all samples
prj.samples
# get fastq file of first sample
prj.samples[0].fastq
# get all bam files of WGBS samples
[s.mapped for s in prj.samples if s.library == "WGBS"]

prj.metadata.results  # results directory of project
# export again the project's annotation
prj.sheet.write(os.path.join(prj.metadata.output_dir, "sample_annotation.csv"))

# project options are read from the config file
# but can be changed on the fly:
prj = Project("test.yaml")
# change options on the fly
prj.config["merge_technical"] = False
```

```
# annotation sheet not specified initially in config file
prj.add_sample_sheet("sample_annotation.csv")
```

class looper.models.**AttributeDict**(*entries=None*, *_force_nulls=False*, *_attribute_identity=False*)
A class to convert a nested mapping(s) into an object(s) with key-values using object syntax (attr_dict.attribute) instead of getitem syntax (attr_dict["key"]). This class recursively sets mappings to objects, facilitating attribute traversal (e.g., attr_dict.attr.attr).

add_entries(*entries*)
Update this *AttributeDict* with provided key-value pairs.

Parameters | collections.Mapping entries (*collections.Iterable*) – collection of pairs of keys and values

copy()
Copy self to a new object.

class looper.models.**PipelineInterface**(*config*)
This class parses, holds, and returns information for a yaml file that specifies how to interact with each individual pipeline. This includes both resources to request for cluster job submission, as well as arguments to be passed from the sample annotation metadata to the pipeline

Parameters **config** (*str | Mapping*) – path to file from which to parse configuration data, or pre-parsed configuration data.

choose_resource_package(*pipeline_name*, *file_size*)
Select resource bundle for given input file size to given pipeline.

Parameters

- **pipeline_name** (*str*) – Name of pipeline.
- **file_size** (*float*) – Size of input data.

Returns resource bundle appropriate for given pipeline, for given input file size

Return type MutableMapping

Raises

- **ValueError** – if indicated file size is negative, or if the file size value specified for any resource package is negative
- **_InvalidResourceSpecificationException** – if no default resource package specification is provided

copy()
Copy self to a new object.

get_arg_string(*pipeline_name*, *sample*, *submission_folder_path=''*, ***null_replacements*)
For a given pipeline and sample, return the argument string

Parameters

- **pipeline_name** (*str*) – Name of pipeline.
- **sample** (Sample) – current sample for which job is being built
- **submission_folder_path** (*str*) – path to folder in which files related to submission of this sample will be placed.
- **null_replacements** (*dict*) – mapping from name of Sample attribute name to value to use in arg string if Sample attribute's value is null

Return str command-line argument string for pipeline

get_attribute(*pipeline_name*, *attribute_key*, *path_as_list=True*)
Return the value of the named attribute for the pipeline indicated.

Parameters

- **pipeline_name** (*str*) – name of the pipeline of interest
- **attribute_key** (*str*) – name of the pipeline attribute of interest
- **path_as_list** (*bool*) – whether to ensure that a string attribute is returned as a list; this is useful for safe iteration over the returned value.

get_pipeline_name(*pipeline*)
Translate a pipeline name (e.g., stripping file extension).

Parameters **pipeline** (*str*) – Pipeline name or script (top-level key in pipeline interface mapping).

Returns translated pipeline name, as specified in config or by stripping the pipeline's file extension

Return type str: translated name for pipeline

pipeline_names
Names of pipelines about which this interface is aware.

Return Iterable[str] names of pipelines about which this interface is aware

pipelines
Keyed collection of pipeline interface data.

Return Mapping pipeline interface configuration data

uses_looper_args(*pipeline_name*)
Determine whether the indicated pipeline uses looper arguments.

Parameters **pipeline_name** (*str*) – name of a pipeline of interest

Returns whether the indicated pipeline uses looper arguments

Return type bool

class looper.models.**Project**(*config_file*, *subproject=None*, *default_compute=None*, *dry=False*, *permissive=True*, *file_checks=False*, *compute_env_file=None*, *no_environment_exception=None*, *no_compute_exception=None*, *defer_sample_construction=False*)
A class to model a Project.

Parameters

- **config_file** (*str*) – Project config file (YAML).
- **subproject** (*str*) – Subproject to use within configuration file, optional
- **default_compute** (*str*) – Configuration file (YAML) for default compute settings.
- **dry** (*bool*) – If dry mode is activated, no directories will be created upon project instantiation.
- **permissive** (*bool*) – Whether a error should be thrown if a sample input file(s) do not exist or cannot be open.
- **file_checks** (*bool*) – Whether sample input files should be checked for their attributes (read type, read length) if this is not set in sample metadata.

- **compute_env_file** (*str*) – Looperenv YAML file specifying compute settings.
- **no_environment_exception** (*type*) – type of exception to raise if environment settings can't be established, optional; if null (the default), a warning message will be logged, and no exception will be raised.
- **no_compute_exception** (*type*) – type of exception to raise if compute settings can't be established, optional; if null (the default), a warning message will be logged, and no exception will be raised.
- **defer_sample_construction** (*bool*) – whether to wait to build this Project's Sample objects until they're needed, optional; by default, the basic Sample is created during Project construction

Example

```
from models import Project
prj = Project("config.yaml")
```

build_sheet(**protocols*)
Create all Sample object for this project for the given protocol(s).

Return pandas.core.frame.DataFrame DataFrame with from base version of each of this Project's samples, for indicated protocol(s) if given, else all of this Project's samples

build_submission_bundles(*protocol*, *priority=True*)
Create pipelines to submit for each sample of a particular protocol.

With the argument (flag) to the priority parameter, there's control over whether to submit pipeline(s) from only one of the project's known pipeline locations with a match for the protocol, or whether to submit pipelines created from all locations with a match for the protocol.

Parameters

- **protocol** (*str*) – name of the protocol/library for which to create pipeline(s)
- **priority** (*bool*) – to only submit pipeline(s) from the first of the pipelines location(s) (indicated in the project config file) that has a match for the given protocol; optional, default True

Return Iterable[(PipelineInterface, str, str)]

Raises AssertionError – if there's a failure in the attempt to partition an interface's pipeline scripts into disjoint subsets of those already mapped and those not yet mapped

compute_env_var
Environment variable through which to access compute settings.

Return str name of the environment variable to pointing to compute settings

copy()
Copy self to a new object.

default_compute_envfile
Path to default compute environment settings file.

finalize_pipelines_directory(*pipe_path=''*)
Finalize the establishment of a path to this project's pipelines.

With the passed argument, override anything already set. Otherwise, prefer path provided in this project's config, then local pipelines folder, then a location set in project environment.

Parameters **pipe_path** (*str*) – (absolute) path to pipelines

Raises

- **PipelinesException** – if (prioritized) search in attempt to confirm or set pipelines directory failed
- **TypeError** – if pipeline(s) path(s) argument is provided and can't be interpreted as a single path or as a flat collection of path(s)

get_arg_string(*pipeline_name*)
For this project, given a pipeline, return an argument string specified in the project config file.

static infer_name(*path_config_file*)
Infer project name based on location of configuration file.

Provide the project with a name, taken to be the name of the folder in which its configuration file lives.

Parameters **path_config_file** (*str*) – path to the project's configuration file.

Return str name of the configuration file's folder, to name project.

make_project_dirs()
Creates project directory structure if it doesn't exist.

num_samples
Number of samples available in this Project.

output_dir
Directory in which to place results and submissions folders.

By default, assume that the project's configuration file specifies an output directory, and that this is therefore available within the project metadata. If that assumption does not hold, though, consider the folder in which the project configuration file lives to be the project's output directory.

Return str path to the project's output directory, either as specified in the configuration file or the folder that contains the project's configuration file.

parse_config_file(*subproject=None*)
Parse provided yaml config file and check required fields exist.

Raises **KeyError** – if config file lacks required section(s)

project_folders
Names of folders to nest within a project output directory.

Return Iterable[str] names of output-nested folders

protocols
Determine this Project's unique protocol names.

Return Set[str] collection of this Project's unique protocol names

required_metadata
Names of metadata fields that must be present for a valid project.

Make a base project as unconstrained as possible by requiring no specific metadata attributes. It's likely that some common-sense requirements may arise in domain-specific client applications, in which case this can be redefined in a subclass.

Return Iterable[str] names of metadata fields required by a project

sample_names
Names of samples of which this Project is aware.

samples
Generic/base Sample instance for each of this Project's samples.

Return Iterable[Sample] Sample instance for each of this Project's samples

set_compute(*setting*)
Set the compute attributes according to the specified settings in the environment file.

Parameters **setting** (*str*) – name for non-resource compute bundle, the name of a subsection in an environment configuration file

Return bool success flag for attempt to establish compute settings

set_project_permissions()
Make the project's public_html folder executable.

templates_folder
Path to folder with default submission templates.

Return str path to folder with default submission templates

update_environment(*env_settings_file*)
Parse data from environment configuration file.

Parameters **env_settings_file** (*str*) – path to file with new environment configuration data

class looper.models.**ProtocolInterface**(*interface_data_source*)
PipelineInterface and ProtocolMapper for a single pipelines location.

This class facilitates use of pipelines from multiple locations by a single project. Also stored are path attributes with information about the location(s) from which the PipelineInterface and ProtocolMapper came.

Parameters **interface_data_source** (*str*) – location (e.g., code repository) of pipelines

fetch_pipelines(*protocol*)
Fetch the mapping for a particular protocol, null if unmapped.

Parameters **protocol** (*str*) – name/key for the protocol for which to fetch the pipeline(s)

Return str | Iterable[str] | NoneType pipeline(s) to which the given protocol is mapped, otherwise null

fetch_sample_subtype(*protocol*, *strict_pipe_key*, *full_pipe_path*)
Determine the interface and Sample subtype for a protocol and pipeline.

Parameters

- **protocol** (*str*) – name of the relevant protocol
- **strict_pipe_key** (*str*) – key for specific pipeline in a pipeline interface mapping declaration; this must exactly match a key in the PipelineInterface (or the Mapping that represent it)
- **full_pipe_path** (*str*) – (absolute, expanded) path to the pipeline script

Return type Sample subtype to use for jobs for the given protocol, that use the pipeline indicated

Raises **KeyError** – if given a pipeline key that's not mapped in this ProtocolInterface instance's PipelineInterface

finalize_pipeline_key_and_paths(*pipeline_key*)
Determine pipeline's full path, arguments, and strict key.

This handles multiple ways in which to refer to a pipeline (by key) within the mapping that contains the data that defines a PipelineInterface. It also ensures proper handling of the path to the pipeline (i.e., ensuring that it's absolute), and that the text for the arguments are appropriately dealt parsed and passed.

Parameters **pipeline_key** (*str*) – the key in the pipeline interface file used for the protocol_mappings section. Previously was the script name.

Return (str, str, str) more precise version of input key, along with absolute path for pipeline script, and full script path + options

class looper.models.**ProtocolMapper**(*mappings_input*)

Map protocol/library name to pipeline key(s). For example, "WGBS" –> wgbs.

Parameters **mappings_input** (*str | Mapping*) – data encoding correspondence between a protocol name and pipeline(s)

build_pipeline(*protocol*)

Create command-line text for given protocol's pipeline(s).

Parameters **protocol** (*str*) – Name of protocol.

copy()

Copy self to a new object.

class looper.models.**Sample**(*series*, *prj=None*)

Class to model Samples based on a pandas Series.

Parameters **series** (*Mapping | pandas.core.series.Series*) – Sample's data.

Example

```
from models import Project, SampleSheet, Sample
prj = Project("ngs")
sheet = SampleSheet("~/projects/example/sheet.csv", prj)
s1 = Sample(sheet.iloc[0])
```

as_series()

Returns a *pandas.Series* object with all the sample's attributes.

Return pandas.core.series.Series pandas Series representation of this Sample, with its attributes.

check_valid(*required=None*)

Check provided sample annotation is valid.

Parameters **required** (*Iterable[str]*) – collection of required sample attribute names, optional; if unspecified, only a name is required.

copy()

Copy self to a new object.

determine_missing_requirements()

Determine which of this Sample's required attributes/files are missing.

Return (type, str) hypothetical exception type along with message about what's missing; null and empty if nothing exceptional is detected

generate_filename(*delimiter='_'*)

Create a name for file in which to represent this Sample.

This uses knowledge of the instance's subtype, sandwiching a delimiter between the name of this Sample and the name of the subtype before the extension. If the instance is a base Sample type, then the filename is simply the sample name with an extension.

Parameters **delimiter** (*str*) – what to place between sample name and name of subtype; this is only relevant if the instance is of a subclass

Return str name for file with which to represent this Sample on disk

generate_name()
Generate name for the sample by joining some of its attribute strings.

get_attr_values(*attrlist*)
Get value corresponding to each given attribute.

Parameters **attrlist** (*str*) – name of an attribute storing a list of attr names

Return list | NoneType value (or empty string) corresponding to each named attribute; null if this Sample's value for the attribute given by the argument to the "attrlist" parameter is empty/null, or if this Sample lacks the indicated attribute

get_sheet_dict()
Create a K-V pairs for items originally passed in via the sample sheet.

This is useful for summarizing; it provides a representation of the sample that excludes things like config files and derived entries.

Return OrderedDict mapping from name to value for data elements originally provided via the sample sheet (i.e., the a map-like representation of the instance, excluding derived items)

infer_columns(*implications*)
Infer value for additional field(s) from other field(s).

Add columns/fields to the sample based on values in those already-set that the sample's project defines as indicative of implications for additional data elements for the sample.

Parameters **implications** (*Mapping*) – Project's implied columns data

Return None this function mutates state and is strictly for effect

is_dormant()
Determine whether this Sample is inactive.

By default, a Sample is regarded as active. That is, if it lacks an indication about activation status, it's assumed to be active. If, however, and there's an indication of such status, it must be '1' in order to be considered switched 'on.'

Return bool whether this Sample's been designated as dormant

locate_data_source(*data_sources*, *column_name='data_source'*, *source_key=None*, *extra_vars=None*)
Uses the template path provided in the project config section "data_sources" to piece together an actual path by substituting variables (encoded by "{variable}"") with sample attributes.

Parameters

- **data_sources** (*Mapping*) – mapping from key name (as a value in a cell of a tabular data structure) to, e.g., filepath
- **column_name** (*str*) – Name of sample attribute (equivalently, sample sheet column) specifying a derived column.
- **source_key** (*str*) – The key of the data_source, used to index into the project config data_sources section. By default, the source key will be taken as the value of the specified column (as a sample attribute). For cases where the sample doesn't have this attribute yet (e.g. in a merge table), you must specify the source key.
- **extra_vars** (*dict*) – By default, this will look to populate the template location using attributes found in the current sample; however, you may also provide a dict of extra variables that can also be used for variable replacement. These extra variables are given a higher priority.

Return str regex expansion of data source specified in configuration, with variable substitutions made

Raises **ValueError** – if argument to data_sources parameter is null/empty

make_sample_dirs()
Creates sample directory structure if it doesn't exist.

set_file_paths(*project*)
Sets the paths of all files for this sample.

Parameters **project** (Project) – object with pointers to data paths and such

set_genome(*genomes*)
Set the genome for this Sample.

Parameters **str] genomes** (*Mapping[str,*) – genome assembly by organism name

set_pipeline_attributes(*pipeline_interface*, *pipeline_name*, *permissive=True*)
Set pipeline-specific sample attributes.

Some sample attributes are relative to a particular pipeline run, like which files should be considered inputs, what is the total input file size for the sample, etc. This function sets these pipeline-specific sample attributes, provided via a PipelineInterface object and the name of a pipeline to select from that interface.

Parameters

- **pipeline_interface** (PipelineInterface) – A PipelineInterface object that has the settings for this given pipeline.
- **pipeline_name** (*str*) – Which pipeline to choose.
- **permissive** (*bool*) – whether to simply log a warning or error message rather than raising an exception if sample file is not found or otherwise cannot be read, default True

set_read_type(*n=10*, *permissive=True*)
For a sample with attr *ngs_inputs* set, this sets the read type (single, paired) and read length of an input file.

Parameters

- **n** (*int*) – Number of reads to read to determine read type. Default=10.
- **permissive** (*bool*) – whether to simply log a warning or error message rather than raising an exception if sample file is not found or otherwise cannot be read, default True

set_transcriptome(*transcriptomes*)
Set the transcriptome for this Sample.

Parameters **str] transcriptomes** (*Mapping[str,*) – transcriptome assembly by organism name

to_yaml(*path=None*, *subs_folder_path=None*, *delimiter='_'*)
Serializes itself in YAML format.

Parameters

- **path** (*str*) – A file path to write yaml to; provide this or the subs_folder_path
- **subs_folder_path** (*str*) – path to folder in which to place file that's being written; provide this or a full filepath
- **delimiter** (*str*) – text to place between the sample name and the suffix within the filename; irrelevant if there's no suffix

Return str filepath used (same as input if given, otherwise the path value that was inferred)

Raises **ValueError** – if neither full filepath nor path to extant parent directory is provided.

update(*newdata*)
Update Sample object with attributes from a dict.

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`). See discussion about this issue here: https://github.com/epigen/looper/issues/8
- **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 directly for flag files for each sample (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.

Changelog

- **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

Support

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

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/epigen/looper

Python Module Index

l

Index

R

S

T

U