pipelines Documentation

Release 0.19.1

Bock lab

Jan 16, 2019

Explanation and Getting Started

peppy is a python package that provides an API for handling standardized project and sample metadata. If you define your project as a standard Portable Encapsulated Project (or PEP), you can use the peppy package to instantiate an in-memory representation of your project and all of its samples. You can then use this for interactive analysis, or to develop a novel python tool based on peppy so you don't have to handle sample processing.

peppy is primarily useful to tool developers and data analysts who want a standard way of representing sample-intensive research project metadata. To get started, proceed with the *Introduction*.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 What are portable encapsulated projects?

A Portable Encapsulated Project (or PEP), is a dataset that subscribes to a standardized structure for organizing metadata. It is written using a simple **yaml + tsv** format that can be read by a variety of tools in the pep toolkit, including *peppy*. If you don't already understand why the PEP concept is useful to you, you should start by reading the explanations on the pepkit website, where you can also find examples of PEP-formatted projects –

1.1.2 What does peppy do?

peppy's job is not to create a PEP for you, but to read it into python and give you an API to interface with that metadata from within python.

1.1.3 Who should use peppy?

There are two key users that will be interested in `peppy`: the python tool developer, and the python data analyst.

Python tool developer. As a tool developer, you should import `peppy` in your python tool and make it so that your tool reads PEP projects as its input. This will make it easy for users to use your tool, because they will already have PEP-formatted projects for other tools.

Python data analyst. `peppy` provides you an easy way to read your project metadata into python. You'll immediately have access to a nice API to interface with your samples and all their attributes, setting the stage for your downstream analysis.

If you don't fit into one of those, you may be interested in the `pepr` R package, which provides an R interface to PEP objects, or the `looper` tool, which lets you run any command-line tool or pipeline on all your samples in your project. Read more about these and other tools at the pepkit website.

1.2 Installing and Hello, World!

1.2.1 Installing

You can install the latest release from pypi using `pip`:

```
pip install --user peppy
```

Update `peppy` with:

```
pip install --user --upgrade peppy
```

Or install any release on the GitHub peppy releases page:

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

1.2.2 Hello world!

Now, to test `peppy`, let's grab an clone an example project that follows PEP format. We've produced a bunch of example PEPs in the example_peps repository on GitHub. Let's clone that repository:

```
git clone https://github.com/pepkit/example_peps.git
```

Then, from within the `example_peps` folder, enter the following commands in a python interactive session:

```
import peppy

proj1 = peppy.Project("example1/project_config.yaml")
samp = proj1.samples
# Find the input file for the first sample in the project
samp[0].file
```

That's it! You've got `peppy` running on an example project. Now you can play around with project metadata from within python. This example and others are explored in more detail in the tutorials section.

1.3 Basic PEP example

The PEP that this example is based on is available in the example_peps repsitory in the example_basic folder.

This vignette will show you a simple example PEP-formatted project, and how to read it into python using the `peppy` package.

Start by importing `peppy`, and then let's take a look at the configuration file that defines our project:

[1]:
```
import peppy
```

[2]:
```
project_config_file = "example_basic/project_config.yaml"
with open(project_config_file) as f:
    print(f.read())
```

```
metadata:
  sample_annotation: sample_annotation.csv
  output_dir: $HOME/hello_looper_results
```

It's a basic `yaml` file with one section, *metadata*, with just two variables. This is about the simplest possible PEP project configuration file. The *sample_annotation* points at the annotation file, which is stored in the same folder as `project_config.yaml`. Let's now glance at that annotation file:

[3]:
```
project_config_file = "example_basic/sample_annotation.csv"
with open(project_config_file) as f:
    print(f.read())
```

```
sample_name,library,file
frog_1,anySampleType,data/frog1_data.txt
frog_2,anySampleType,data/frog2_data.txt
```

This *sample_annotation* file is a basic *csv* file, with rows corresponding to samples, and columns corresponding to sample attributes. Let's read this simple example project into python using `peppy`:

[4]:
```
proj = peppy.Project("example_basic/project_config.yaml")
```

Now, we have access to all the project metadata in easy-to-use form using python objects. We can browse the samples in the project like this:

[5]:
```
proj.samples[0].file
```

[5]:
```
'data/frog1_data.txt'
```

[ ]:
```

```

1.4 Sample subannotation

The PEPs that this examples are based on are available in the example_peps repsitory.

This vignette will show you how sample subannotations work in a series of examples.

Import libraries and set the working directory

[1]:
```
import os
import peppy
os.chdir("/Users/mstolarczyk/Uczelnia/UVA/")
```

1.4.1 Example 1: basic sample subannotation table

Example 1 demonstrates how a `sample_subannotation` is used. In this example, 2 samples have multiple input files that need merging (`frog_1` and `frog_2`), while 1 sample (`frog_3`) does not. Therefore, `frog_3` specifies its file in the `sample_annotation` table, while the others leave that field blank and instead specify several files in the `sample_subannotation`.

[2]:
```
p1 = peppy.Project("example_peps/example_subannotation1/project_config.yaml")
p1.samples[0].file
```

[2]:
```
'data/frog1a_data.txt data/frog1b_data.txt data/frog1c_data.txt'
```

[3]:
```
ss = p1.get_subsample(sample_name="frog_1", subsample_name="sub_a")
print(type(ss))
print(ss)
```

```
<class 'peppy.sample.Subsample'>
Subsample: {'sample_name': 'frog_1', 'subsample_name': 'sub_a', 'file': 'data/
↪frog1a_data.txt'}
```

1.4.2 Example 2: subannotations and derived columns

Example 2 uses a `sample_subannotation` table and a derived column to point to files. This is a rather complex example. Notice we must include the `file_id` column in the `sample_annotation` table, and leave it blank; this is then populated by just some of the samples (`frog_1` and `frog_2`) in the `sample_subannotation`, but is left empty for the samples that are not merged.

[4]:
```
import peppy
p2 = peppy.Project("example_peps/example_subannotation2/project_config.yaml")
p2.samples[0].file
```

[4]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../data/frog1a_
↪data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../
↪data/frog1b_data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_
↪subannotation2/../data/frog1c_data.txt'
```

[5]:
```
p2.samples[1].file
```

[5]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../data/frog2a_
↪data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../
↪data/frog2b_data.txt'
```

[6]:
```
p2.samples[2].file
```

[6]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../data/frog3_
↪data.txt'
```

[7]:
```
p2.samples[3].file
```

[7]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation2/../data/frog4_
↪data.txt'
```

1.4.3 Example 3: subannotations and expansion characters

This example gives the exact same results as example 2, but in this case, uses a wildcard for `frog_2` instead of including it in the `sample_subannotation` table. Since we can't use a wildcard and a subannotation for the same sample, this necessitates specifying a second data source class (`local_files_unmerged`) that uses an asterisk. The outcome is the same.

[8]:
```
p3 = peppy.Project("example_peps/example_subannotation3/project_config.yaml")
p3.samples[0].file
```

[8]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../data/frog1a_
↪data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../
↪data/frog1b_data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_
↪subannotation3/../data/frog1c_data.txt'
```

[9]:
```
p3.samples[1].file
```

[9]:
```
'/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../data/frog2_
↪data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../
↪data/frog2a_data.txt /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_
↪subannotation3/../data/frog2b_data.txt'
```

(continues on next page)

(continued from previous page)

```
[10]: p3.samples[2].file
[10]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../data/frog3_
      ↪data.txt'

[11]: p3.samples[3].file
[11]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation3/../data/frog4_
      ↪data.txt'
```

1.4.4 Example 4: subannotations and multiple (separate-class) inputs

Merging is for same class inputs (like, multiple files for read1). Different-class inputs (like read1 vs read2) are handled by different attributes (or columns). This example shows you how to handle paired-end data, while also merging within each.

```
[12]: p4 = peppy.Project("example_peps/example_subannotation4/project_config.yaml")
      p4.samples[0].read1
[12]: 'frog1a_data.txt frog1b_data.txt frog1c_data.txt'

[13]: p4.samples[0].read2
[13]: 'frog1a_data2.txt frog1b_data2.txt frog1b_data2.txt'
```

1.4.5 Example 5: subannotations and multiple (separate-class) inputs with derived columns

Merging is for same class inputs (like, multiple files for read1). Different-class inputs (like read1 vs read2) are handled by different attributes (or columns). This example shows you how to handle paired-end data, while also merging within each.

```
[14]: p5 = peppy.Project("example_peps/example_subannotation5/project_config.yaml")
      p5.samples[0].read1
[14]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog1a_
      ↪R1.fq.gz /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../
      ↪data/frog1b_R1.fq.gz /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_
      ↪subannotation5/../data/frog1c_R1.fq.gz'

[15]: p5.samples[0].read2
[15]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog1a_
      ↪R2.fq.gz /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../
      ↪data/frog1b_R2.fq.gz /Users/mstolarczyk/Uczelnia/UVA/example_peps/example_
      ↪subannotation5/../data/frog1c_R2.fq.gz'

[16]: p5.samples[1].read1
[16]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog2_R1.
      ↪fq.gz'
```

```
[17]: p5.samples[1].read2
[17]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog2_R2.
      ↪fq.gz'

[18]: p5.samples[2].read1
[18]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog3_R1.
      ↪fq.gz'

[19]: p5.samples[2].read2
[19]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog3_R2.
      ↪fq.gz'

[20]: p5.samples[3].read1
[20]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog4_R1.
      ↪fq.gz'

[21]: p5.samples[3].read2
[21]: '/Users/mstolarczyk/Uczelnia/UVA/example_peps/example_subannotation5/../data/frog4_R2.
      ↪fq.gz'
```

1.5 Project models

`peppy` models projects and samples as objects in Python.

```
import peppy

my_project = peppy.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. 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.

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.

1.6 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, project=None):
                """Sets the paths of all files for this sample."""
```

(continues on next page)

(continued from previous page)

```
                # Inherit paths from Sample by running Sample's set_file_paths()
                super(ATACseqSample, self).set_file_paths(project)

                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
        ...
        ...
    ...
    ...
```

1.7 API

1.7.1 peppy

Project configuration, particularly for logging.

Project-scope constants may reside here, but more importantly, some setup here will provide a logging infrastructure for all of the project's modules. Individual modules and classes may provide separate configuration on a more local level, but this will at least provide a foundation.

class peppy.**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 **object)] | Mapping | pandas.Series entries** (*Iterable[(object,)*) – collection of pairs of keys and values

Return AttributeDict the updated instance

copy()

Copy self to a new object.

is_null(*item*)

Conjunction of presence in underlying mapping and value being None

Parameters **item** (*object*) – Key to check for presence and null value

Return bool True iff the item is present and has null value

non_null(*item*)

Conjunction of presence in underlying mapping and value not being None

Parameters **item** (*object*) – Key to check for presence and non-null value

Return bool True iff the item is present and has non-null value

class peppy.**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 (collection of samples and metadata).

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*) – Environment configuration 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")
```

exception MissingMetadataException(*missing_section*, *path_config_file=None*)

Project needs certain metadata.

exception MissingSampleSheetError(*sheetfile*)

Represent case in which sample sheet is specified but nonexistent.

activate_subproject(*subproject*)

Activate a subproject.

This method will update Project attributes, adding new values associated with the subproject indicated, and in case of collision with an existing key/attribute the subproject's value will be favored.

Parameters **subproject** (*str*) – A string with a subproject name to be activated

Return Project A Project with the selected subproject activated

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

compute_env_var
Environment variable through which to access compute settings.

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

constants
Return key-value pairs of pan-Sample constants for this Project.

Return Mapping collection of KV pairs, each representing a pairing of attribute name and attribute value

copy()
Copy self to a new object.

default_compute_envfile
Path to default compute environment settings file.

Return str Path to this project's default compute env config file.

derived_columns
Collection of sample attributes for which value of each is derived from elsewhere

Return list[str] sample attribute names for which value is derived

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.

get_sample(*sample_name*)
Get an individual sample object from the project.

Will raise a ValueError if the sample is not found. In the case of multiple samples with the same name (which is not typically allowed), a warning is raised and the first sample is returned.

Parameters **sample_name** (*str*) – The name of a sample to retrieve

Return Sample The requested Sample object

get_samples(*sample_names*)
Returns a list of sample objects given a list of sample names

Parameters **sample_names** (*list*) – A list of sample names to retrieve

Return list[Sample] A list of Sample objects

implied_columns
Collection of sample attributes for which value of each is implied by other(s)

Return list[str] sample attribute names for which value is implied by other(s)

infer_name()
Infer project name from config file path.

First assume the name is the folder in which the config file resides, unless that folder is named "metadata", in which case the project name is the parent of that folder.

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

Return str inferred name for project.

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

num_samples
Count the number of samples available in this Project.

Return int 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.

Parameters **subproject** (*str*) – Name of subproject to activate, optional

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

static parse_sample_sheet(*sample_file*, *dtype=<type 'str'>*)
Check if csv file exists and has all required columns.

Parameters

- **sample_file** (*str*) – path to sample annotations file.
- **dtype** (*type*) – data type for CSV read.

Raises

- **IOError** – if given annotations file can't be read.
- **ValueError** – if required column(s) is/are missing.

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.

sheet
Annotations/metadata sheet describing this Project's samples.

Return pandas.core.frame.DataFrame table of samples in this Project

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 peppy.**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.

Return (Exception | NoneType, str, str) exception and messages about what's missing/empty; null with empty messages if there was nothing exceptional or required inputs are absent or not set

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)

get_subsample(*subsample_name*)
Retrieve a single subsample by name.

Parameters **subsample_name** (*str*) – The name of the desired subsample. Should match the subsample_name column in the subannotation sheet.

Return Subsample Requested Subsample object

get_subsamples(*subsample_names*)
Retrieve subsamples assigned to this sample

Parameters **subsample_names** (*list*) – List of names of subsamples to retrieve

Return list List of subsamples

infer_attributes(*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

input_file_paths
List the sample's data source / input files

Return list[str] paths to data sources / input file for this Sample.

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

library
Backwards-compatible alias.

Return str The protocol / NGS library name for this Sample.

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=None*)
Sets the paths of all files for this sample.

Parameters **project** (AttributeDict) – object with pointers to data paths and such, either full Project or AttributeDict with sufficient data

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 (*rlen_sample_size=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

- **rlen_sample_size** (*int*) – Number of reads to sample to infer 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*, ***kwargs*)
Update Sample object with attributes from a dict.

exception peppy.**PeppyError** (*msg*)
Base error type for peppy custom errors.

1.8 Changelog

- **v0.19** (*2019-01-16*):
  - Changed

        * `Project` construction no longer requires sample annotations sheet.
        * Specification of assembly/ies in project config outside of `implied_attributes` is deprecated.
        * `implied_columns` and `derived_columns` are deprecated in favor of `implied_attributes` and `derived_attributes`.
    - New
        * Added `activate_subproject` method to `Project`.
- **v0.18.2** (*2018-07-23*):
    - Fixed
        * Made requirements more lenient to allow for newer versions of required packages.
- **v0.18.1** (*2018-06-29*):
    - Fixed
        * Fixed a bug that would cause sample attributes to lose order.
        * Fixed a bug that caused an install error with newer `numexpr` versions.
    - New
        * Project names are now inferred with the `infer_name` function, which uses a priority lookup to infer the project name: First, the `name` attribute in the `yaml` file; otherwise, the containing folder unless it is `metadata`, in which case, it's the parent of that folder.
        * Add `get_sample` and `get_samples` functions to `Project` objects.
        * Add `get_subsamples` and `get_subsample` functions to both `Project` and `Sample` objects.
        * Subsamples are now objects that can be retrieved individually by name, with the `subsample_name` as the index column header.
- **v0.17.2** (*2018-04-03*):
    - Fixed
        * Ensure data source path relativity is with respect to project config file's folder.
- **v0.17.1** (*2017-12-21*):
    - Changed
        * Version bump for first pypi release
        * Fixed bug with packaging for pypi release
- **v0.9** (*2017-12-21*):
    - New
        * Separation completed, `peppy` package is now standalone
        * `looper` can now rely on `peppy`
    - Changed
        * `merge_table` renamed to `sample_subannotation`
        * setup changed for compatibility with Pypi
- **v0.8.1** (*2017-11-16*):
    - New

        * Separated from looper into its own python package (originally called *pep*).

- **v0.7.2** (*2017-11-16*):
  - Fixed
    * Correctly count successful command submissions when not using *–dry-run*.
- **v0.7.1** (*2017-11-15*):
  - Fixed
    * No longer falsely display that there's a submission failure.
    * Allow non-string values to be unquoted in the `pipeline_args` section.
- **v0.7** (*2017-11-15*):
  - New
    * Add `--lump` and `--lumpn` options
    * Catch submission errors from cluster resource managers
    * Implied columns can now be derived
    * Now protocols can be specified on the command-line *–include-protocols*
    * Add rudimentary figure summaries
    * Simplifies command-line help display
    * Allow wildcard protocol_mapping for catch-all pipeline assignment
    * Improve user messages
    * New sample_subtypes section in pipeline_interface
  - Changed
    * Sample child classes are now defined explicitly in the pipeline interface. Previously, they were guessed based on presence of a class extending Sample in a pipeline script.
    * Changed 'library' key sample attribute to 'protocol'
- **v0.6** (*2017-07-21*):
  - New
    * Add support for implied_column section of the project config file
    * Add support for Python 3
    * Merges pipeline interface and protocol mappings. This means we now allow direct pointers to `pipeline_interface.yaml` files, increasing flexibility, so this relaxes the specified folder structure that was previously used for `pipelines_dir` (with `config` subfolder).
    * Allow URLs as paths to sample sheets.
    * Allow tsv format for sample sheets.
    * Checks that the path to a pipeline actually exists before writing the submission script.
  - Changed
    * Changed LOOPERENV environment variable to PEPENV, generalizing it to generic models
    * Changed name of `pipelines_dir` to `pipeline_interfaces` (but maintained backwards compatibility for now).

    * Changed name of `run` column to `toggle`, since `run` can also refer to a sequencing run.
    * Relaxes many constraints (like resources sections, pipelines_dir columns), making project configuration files useful outside looper. This moves us closer to dividing models from looper, and improves flexibility.
    * Various small bug fixes and dev improvements.
    * Require *setuptools* for installation, and *pandas 0.20.2*. If *numexpr* is installed, version *2.6.2* is required.
    * Allows tilde in `pipeline_interfaces`

- **v0.5** (*2017-03-01*):
  - New
    * Add new looper version tracking, with *–version* and *-V* options and printing version at runtime
    * Add support for asterisks in file paths
    * Add support for multiple pipeline directories in priority order
    * Revamp of messages make more intuitive output
    * Colorize output
    * Complete rehaul of logging and test infrastructure, using logging and pytest packages
  - Changed
    * Removes pipelines_dir requirement for models, making it useful outside looper
    * Small bug fixes related to *all_input_files* and *required_input_files* attributes
    * More robust installation and more explicit requirement of Python 2.7
- **v0.4** (*2017-01-12*):
  - New
    * New command-line interface (CLI) based on sub-commands
    * New subcommand (`looper summarize`) replacing the `summarizePipelineStats.R` script
    * New subcommand (`looper check`) replacing the `flagCheck.sh` script
    * New command (`looper destroy`) to remove all output of a project
    * New command (`looper clean`) to remove intermediate files of a project flagged for deletion
    * Support for portable and pipeline-independent allocation of computing resources with Looperenv.
  - Changed
    * Removed requirement to have `pipelines` repository installed in order to extend base Sample objects
    * Maintenance of sample attributes as provided by user by means of reading them in as strings (to be improved further)
    * Improved serialization of Sample objects

1.9 Support

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

1.10 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

p

Index

T

U