Pimlico Documentation

Release 0.9.25

Mark Granroth-Wilding

Dec 17, 2020

Contents

The **Pimlico Processing Toolkit** is a toolkit for building pipelines of tasks for **processing large datasets** (corpora). It is especially focussed on processing linguistic corpora and provides wrappers around many existing, widely used **NLP** (Natural Language Processing) tools.

It makes it easy to write large, potentially complex pipelines with the key goals of making is easy to:

- provide **clear documentation** of what has been done;
- **incorporate standard NLP tasks** and data-processing tasks with minimal effort;
- **integrate non-standard code**, specific to the task at hand, into the same pipeline; and
- **distribute code** for later reproduction or application to other datasets or experiments.

The toolkit takes care of managing data between the steps of a pipeline and checking that everything's executed in the right order.

The core toolkit is written in Python. Pimlico is open source, released under the GPLv3 license. It is available from its Github repository.

- For a broad introduction to Pimlico's key concepts, read *Introduction to Pimlico*.
- To get started with a Pimlico project, follow the *getting-started guide*.

Pimlico is written in Python and can be run using Python >=2.7 or >=3.6. This means you can write your own processing modules using either Python 2 or 3.

Pimlico is short for *PIpelined Modular LInguistic COrpus processing*.

CHAPTER 1

Contents

1.1 Pimlico guides

Step-by-step guides through common tasks while using Pimlico.

1.1.1 Introduction to Pimlico

Motivation

It is becoming more and more common for conferences and journals in NLP and other computational areas to encourage, or even require, authors to make publicly available the code and data required to reproduce their reported results. It is now widely acknowledged that such practices lie at the center of open science and are essential to ensuring that research contributions are verifiable, extensible and useable in applications.

However, this requires extensive additional work. And, even when researchers do this, it is all too common for others to have to spend large amounts of time and effort preparing data, downloading and installing tools, configuring execution environments and picking through instructions and scripts before they can reproduce the original results, never mind apply the code to new datasets or build upon it in novel research.

Introducing Pimlico

Pimlico (**Pi**pelined **M**odular **Li**nguistic **Co**rpus processing) addresses these problems. It allows users to write and run potentially complex processing pipelines, with the key goals of making it easy to:

- clearly document what was done;
- incorporate standard NLP and data-processing tasks with minimal effort;
- integrate non-standard code, specific to the task at hand, into the same pipeline; and
- distribute code for later reproduction or application to other datasets or experiments.

It comes with pre-defined **module types** to wrap a number of existing **NLP toolkits** (including non-Python code) and carry out many other common pre-processing or data manipulation tasks.

Building pipelines

Pimlico addresses the task of **building of pipelines to process large datasets**. It allows you to run one or several steps of processing at a time, with high-level control over how each step is run, manages the data produced by each step, and lets you observe these intermediate outputs. Pimlico provides simple, powerful tools to give this kind of control, without needing to write any code.

Developing a pipeline with Pimlico involves defining the **structure of the pipeline** itself in terms of **modules** to be executed, and **connections between their inputs and outputs** describing the flow of data.

Modules correspond to some data-processing code, with some parameters. They may be of a standard type, so-called **core module types**, for which code is provided as part of Pimlico.

A pipeline may also incorporate **custom module types**, for which metadata and data-processing code must be provided by the author.

Pipeline configuration

> See *Pipeline config* for more on pipeline configuration.

At the heart of Pimlico is the concept of a **pipeline configuration**, defined by a configuration (or *conf*) file, which can be loaded and executed.

This specifies some general parameters and metadata regarding the pipeline and then a sequence of modules to be executed.

Each **pipeline module** is defined by a named section in the file, which specifies the module type, inputs to be read from the outputs of other, previous modules, and parameters.

For example, the following configuration section defines a module called `split`. Its type is the core Pimlico module type *`corpus split`*, which splits a corpus by documents into two randomly sampled subsets (as is typically done to produce training and test sets).

```
[split]
type=pimlico.modules.corpora.split
input=tokenized_corpus
set1_size=0.8
```

The option `input` specifies where the module's only input comes from and refers by name to a module defined earlier in the pipeline whose output provides the data. The option `set1_size` tells the module to put 80% of documents into the first set and 20% in the second. Two outputs are produced, which can be referred to later in the pipeline as `split.set1` and `split.set2`.

Input modules

The first module(s) of a pipeline have no inputs, but load datasets, with parameters to specify where the input data can be found on the filesystem.

A number of *`standard input readers`* are among Pimlico's core module types to support reading of simple datasets, such as text files in a directory, and some standard input formats for data such as word embeddings. The toolkit also provides a factory to make it easy to define custom routines for reading other types of input data.

Module type

The **type** of a module is given as a fully qualified Python path to a Python package.

The package provides separately the module type's metadata, referred to as its *module info* – input datatypes, options, etc. – and the code that is executed when it is run, the *module executor*. The example above uses one of Pimlico's core module types.

A pipeline will usually also include non-standard module types, distributed together with the conf file. These are defined and used in exactly the same way as the core module types. Where custom module types are used, the pipeline conf file specifies a directory where the source code can be found.

See also:

Full worked example

An example of a complete pipeline conf, using both core and custom module types

Datatypes

When a module is run, its output is stored ready for use by subsequent modules. Pimlico takes care of storing each module's output in separate locations and providing the correct data as input.

The module info for a module type defines a **datatype** for each input and each output. Pimlico includes a system of datatypes for the datasets that are passed between modules.

When a pipeline is loaded, type-checking is performed on the connections between modules' outputs and subsequent modules' inputs to ensure that appropriate datatypes are provided.

For example, a module may require as one of its inputs a vocabulary, for which Pimlico provides a `standard datatype`. The pipeline will only be loaded if this input is connected to an output that supplies a compatible type. The supplying module does not need to define how to store a vocabulary, since the datatype defines the necessary routines for **writing a vocabulary to disk**. The subsequent module does not need to define how to **read the data** either, since the datatype takes care of that too, providing the module executor with suitable Python data structures.

Corpora

Often modules read and write **corpora**, consisting of a large number of documents. Pimlico provides a datatype for representing such corpora and a further type system for the **types of the documents** stored within a corpus (rather like Java's *generic* types).

For example, a module may specify that it requires as input a corpus whose documents contain tokenized text. All tokenizer modules (of which there are several) provide output corpora with this document type. The corpus datatype takes care of reading and writing large corpora, preserving the order of documents, storing corpus metadata, and much more.

The datatype system is also extensible in custom code. As well as defining custom module types, a pipeline author may wish to define new datatypes to represent the data required as input to the modules or provided as output.

See also:

`IterableCorpus`: datatype for corpora.

Running a pipeline

Pimlico provides a command-line interface for parsing and executing pipelines. The interface provides sub-commands to perform different operations relating to a given pipeline. The conf file defining the pipeline is always given as an argument and the first operation is therefore to parse the pipeline and check it for validity. We describe here a few of the most important sub-commands.

See also:

Command-line interface

A complete list of the available commands

status

The status subcommand

Outputs a list of all of the modules in the pipeline, reporting the execution status of each. This indicates whether the module has been run; if so, whether it completed successfully or failed; if not, whether it is ready to be run (i.e. all of its input data is available).

Each of the modules is numbered in the list, and this number can be used instead of the module's full name in arguments to all sub-commands.

Given the name of a module, the command outputs a detailed report on the status of that module and its input and output datasets.

run

The run subcommand

Executes a module.

An option `--dry` runs all pre-execution checks for the module, without running it. These include checking that required software is installed and performing automatic installation if not.

If all requirements are satisfied, the module will be executed, outputting its progress to the terminal and to module-specific log files. Output datasets are written to module-specific directories, ready to be used by subsequent modules later.

Multiple modules can be run in sequence, or even the entire pipeline. A switch `--all-deps` causes any unexecuted modules upon whose output the specified module(s) depend to be run.

browse

The browse subcommand

Inspects the data output by a module, stored in its pipeline-internal storage. Inspecting output data by loading the files output by the module would require knowledge of both the Pimlico data storage system and the specific storage formats used by the output datatypes. Instead, this command lets the user inspect the data from a given module (and a given output, if there are multiple).

Datatypes, as part of their definition, along with specification of storage format reading and writing, define how the data can be formatted for display. Multiple formatters may be defined, giving alternative ways to inspect the same data.

For some datatypes, browsing is as simple as outputting some statistics about the data, or a string representing its contents. For corpora, a document-by-document browser is provided, using the Urwid library. Furthermore, the definition of corpus document types determines how an individual document should be displayed in the corpus browser. For example, the tokenized text type shows each sentence on a separate line, with spaces between tokens.

Where next?

For a practical quick-start guide to building pipelines, see *Super-quick Pimlico setup*.

Or for a bit more detail, see *Setting up a new project using Pimlico*.

1.1.2 Super-quick Pimlico setup

This is a very quick walk-through of the process of starting a new project using Pimlico. For more details, explanations, etc see *the longer getting-started guide*.

First, make sure Python is installed.

System-wide configuration

Choose a location on your file system where Pimlico will store all the output from pipeline modules. For example, `/home/me/.pimlico_store/`.

Create a file in your home directory called `.pimlico` that looks like this:

```
store=/home/me/.pimlico_store
```

This is not specific to a pipeline: separate pipelines use separate subdirectories.

Set up new project

Create a new, empty directory to put your project in. E.g.:

```
cd ~
mkdir myproject
```

Download newproject.py into this directory and run it:

```
wget https://raw.githubusercontent.com/markgw/pimlico/master/admin/newproject.py
python newproject.py myproject
```

This fetches the latest Pimlico codebase (in `pimlico/`) and creates a template pipeline (`myproject.conf`).

Customizing the pipeline

You've got a basic pipeline config file now (`myproject.conf`).

Add sections to it to configure modules that make up your pipeline.

For guides to doing that, see the *the longer setup guide* and individual module documentation.

Running Pimlico

Check the pipeline can be loaded and take a look at the list of modules you've configured:

```
./pimlico.sh myproject.conf status
```

Tell the modules to fetch all the dependencies you need:

```
./pimlico.sh myproject.conf install all
```

If there's anything that can't be installed automatically, this should output instructions for manual installation.

Check the pipeline's ready to run a module that you want to run:

```
./pimlico.sh myproject.conf run MODULE --dry-run
```

To run the next unexecuted module in the list, use:

```
./pimlico.sh myproject.conf run
```

1.1.3 Setting up a new project using Pimlico

You've decided to use Pimlico to implement a data processing pipeline. So, where do you start?

This guide steps through the basic setup of your project. You don't have to do everything exactly as suggested here, but it's a good starting point and follows Pimlico's recommended procedures. It steps through the setup for a very basic pipeline.

A shorter version of this guide that zooms through the essential setup steps is also available.

System-wide configuration

Note: If you've used Pimlico before, you can skip this step.

Pimlico needs you to specify certain parameters regarding your local system. Typically this is just a file in your home directory called `.pimlico`. *More details*.

It needs to know where to put output files as it executes. These settings apply to all Pimlico pipelines you run. Pimlico will make sure that different pipelines don't interfere with each other's output (provided you give them different names).

Most of the time, you only need to specify one storage location, using the `store` parameter in your local config file. (You can specify multiple: *more details*.)

Create a file `~/.pimlico` that looks like this:

```
store=/path/to/storage/directory
```

All pipelines will use different subdirectories of this one.

Getting started with Pimlico

The procedure for starting a new Pimlico project, using the latest release, is very simple.

Create a new, empty directory to put your project in. Download newproject.py into the project directory.

Make sure you've got Python installed. Pimlico currently supports Python 2 and 3, but we strongly recommend using Python 3 unless you have old Python 2 code you need to run.

Choose a name for your project (e.g. `myproject`) and run:

```
python newproject.py myproject
```

This fetches the latest version of Pimlico (now in the `pimlico/` subdirectory) and creates a basic config file, which will define your pipeline.

It also retrieves libraries that Pimlico needs to run. Other libraries required by specific pipeline modules will be installed as necessary when you use the modules.

Building the pipeline

You've now got a config file in `myproject.conf`. This already includes a `pipeline` section, which gives the basic pipeline setup. It will look something like this:

```
[pipeline]
name=myproject
release=<release number>
python_path=%(project_root)s/src/python
```

The `name` needs to be distinct from any other pipelines that you run – it's what distinguishes the storage locations.

`release` is the release of Pimlico that you're using: it's automatically set to the latest one, which has been downloaded.

If you later try running the same pipeline with an updated version of Pimlico, it will work fine as long as it's the same minor version (the second part). The minor-minor third part can be updated and may bring some improvements. If you use a higher minor version (e.g. 0.10.x when you started with 0.9.24), there may be backwards incompatible changes, so you'd need to update your config file, ensuring it plays nicely with the later Pimlico version.

Getting input

Now we add our first module to the pipeline. This reads input from a collection of text files. We use a small subset of the Europarl corpus as an example here. This can be simply adapted to reading the real Europarl corpus or any other corpus stored in this straightforward way.

Download and extract the small corpus from here

In the example below, we have extracted the files to a directory `data/europarl_demo` in the home directory.

```
[input_text]
type=pimlico.modules.input.text.raw_text_files
files=%(home)s/data/europarl_demo/*
```

Doing something: tokenization

Now, some actual linguistic processing, albeit somewhat uninteresting. Many NLP tools assume that their input has been divided into sentences and tokenized. To keep things simple, we use a very basic, regular expression-based tokenizer.

Notice that the output from the previous module feeds into the input for this one, which we specify simply by naming the module.

```
[tokenize]
type=pimlico.modules.text.simple_tokenize
input=input_text
```

Doing something more interesting: POS tagging

Many NLP tools rely on part-of-speech (POS) tagging. Again, we use OpenNLP, and a standard Pimlico module wraps the OpenNLP tool.

```
[pos-tag]
type=pimlico.modules.opennlp.pos
input=tokenize
```

Running Pimlico

Now we've got our basic config file ready to go. It's a simple linear pipeline that goes like this:

> read input docs -> group into batches -> tokenize -> POS tag

It's now ready to load and inspect using Pimlico's command-line interface.

Before we can run it, there's one thing missing: the OpenNLP tokenizer module needs access to the OpenNLP tool. We'll see below how Pimlico sorts that out for you.

Checking everything's dandy

Now you can run the `status` command to check that the pipeline can be loaded and see the list of modules.

```
./pimlico.sh myproject.conf status
```

To check that specific modules are ready to run, with all software dependencies installed, use the `run` command with `--dry-run` (or `--dry`) switch:

```
./pimlico.sh myproject.conf run tokenize --dry
```

Fetching dependencies

All the standard modules provide easy ways to get hold of their dependencies automatically, or as close as possible. Most of the time, all you need to do is tell Pimlico to install them.

You use the `run` command, with a module name and `--dry-run`, to check whether a module is ready to run.

```
./pimlico.sh myproject.conf run tokenize --dry
```

This will find that things aren't quite ready yet, as the OpenNLP Java packages are not available. These are not distributed with Pimlico, since they're only needed if you use an OpenNLP module.

When you run the `run` command, Pimlico will offer to install the necessary software for you. In this case, this involves downloading OpenNLP's jar files from its web repository to somewhere where the OpenNLP tokenizer module can find it.

Say yes and Pimlico will get everything ready. Simple as that!

There's one more thing to do: the tools we're using require statistical models. We can simply download the pre-trained English models from the OpenNLP website.

At present, Pimlico doesn't yet provide a built-in way for the modules to do this, as it does with software libraries, but it does include a GNU Makefile to make it easy to do:

```
cd ~/myproject/pimlico/models
make opennlp
```

Note that the modules we're using default to these standard, pre-trained models, which you're now in a position to use. However, if you want to use different models, e.g. for other languages or domains, you can specify them using extra options in the module definition in your config file.

If there are any other library problems shown up by the dry run, you'll need to address them before going any further.

Running the pipeline

What modules to run?

Pimlico suggests an order in which to run your modules. In our case, this is pretty obvious, seeing as our pipeline is entirely linear – it's clear which ones need to be run before others.

```
./pimlico.sh myproject.conf status
```

The output also tells you the current status of each module. At the moment, all the modules are `UNEXECUTED`.

You might be surprised to see that `input-text` features in the list. This is because, although it just reads the data out of a corpus on disk, there's not quite enough information in the corpus, so we need to run the module to collect a little bit of metadata from an initial pass over the corpus. Some input types need this, others not. In this case, all we're lacking is a count of the total number of documents in the corpus.

Note: To make running your pipeline even simpler, you can abbreviate the command by using a **shebang** in the config file. Add a line at the top of `myproject.conf` like this:

```
#!./pimlico.sh
```

Then make the conf file executable by running (on Linux):

```
chmod ug+x myproject.conf
```

Now you can run Pimlico for your pipeline by using the config file as an executable command:

```
./myproject.conf status
```

Running the modules

The modules can be run using the `run` command and specifying the module by name. We do this manually for each module.

```
./pimlico.sh myproject.conf run input-text
./pimlico.sh myproject.conf run tokenize
./pimlico.sh myproject.conf run pos-tag
```

Adding custom modules

Most likely, for your project you need to do some processing not covered by the built-in Pimlico modules. At this point, you can start implementing your own modules, which you can distribute along with the config file so that people can replicate what you did.

The `newproject.py` script has already created a directory where our custom source code will live: `src/python`, with some subdirectories according to the standard code layout, with module types and datatypes in separate packages.

The template pipeline also already has an option `python_path` pointing to this directory, so that Pimlico knows where to find your code. Note that the code's in a subdirectory of that containing the pipeline config and we specify the custom code path relative to the config file, so it's easy to distribute the two together.

Now you can create Python modules or packages in `src/python`, following the same conventions as the built-in modules and overriding the standard base classes, as they do. The following articles tell you more about how to do this:

- *Writing Pimlico module types*
- *Writing document map modules*
- *Pimlico module structure*

Your custom modules and datatypes can then simply be used in the config file as module types.

1.1.4 Running someone else's pipeline

This guide takes you through what to do if you have received someone else's code for a Pimlico project and would like to run it.

This guide is written for Unix/Mac users. You'll need to make some adjustments if using another OS.

What you've got

Hopefully got at least a pipeline config file. This will have the extension `.conf`. In the examples below, we'll use the name `myproject.conf`.

You've probably got a whole directory, with some subdirectories, containing this config file (or even several) together with other related files – datasets, code, etc. This top-level directory is what we'll refer to as the *project root*.

The project may include some code, probably defining some custom Pimlico module types and datatypes. If all is well, you won't need to delve into this, as its location will be given in the config file and Pimlico will take care of the rest.

Getting Pimlico

You hopefully didn't receive the whole Pimlico codebase together with the pipeline and code. It's recommended not to distribute Pimlico, as it can be fetched automatically for a given pipeline.

You'll need Python installed.

Download the Pimlico bootstrap script from here and put it in the project root.

Now run it:

```
python bootstrap.py myproject.conf
```

The bootstrap script will look in the config file to work out what version of Pimlico to use and then download it.

If this works, you should now be able to run Pimlico.

Using the bleeding edge code

By default, the bootstrap script will fetch a release of Pimlico that the config file declares as being that which it was built with.

If you want the very latest version of Pimlico, with all the dangers that entails and with the caveat that it might not work with the pipeline you're trying to run, you can tell the bootstrap script to checkout Pimlico from its Git repository.

```
python bootstrap.py --git myproject.conf
```

Running Pimlico

Perhaps the project root contains a (link to a) script called `pimlico.sh`.

If not, create one like this:

```
ln -s pimlico/bin/pimlico.sh .
```

Now run `pimlico.sh` with the config file as an argument, issuing the command_status command to see the contents of the pipeline:

```
./pimlico.sh myproject.conf status
```

Pimlico will now run and set itself up, before proceeding with your command and showing the pipeline status. This might take a bit of time. It will install a Python virtual environment and some basic packages needed for it to run.

1.1.5 Writing Pimlico module types

Pimlico comes with a fairly large number of *module types* that you can use to run many standard NLP, data processing and ML tools over your datasets.

For some projects, this is all you need to do. However, often you'll want to mix standard tools with your own code, for example, using the output from the tools. And, of course, there are many more tools you might want to run that aren't built into Pimlico: you can still benefit from Pimlico's framework for data handling, config files and so on.

For a detailed description of the structure of a Pimlico module, see *Pimlico module structure*. This guide takes you through building a simple module.

Note: In any case where a module will process a corpus one document at a time, you should write a *document map module*, which takes care of a lot of things for you, so you only need to say what to do with each document.

Todo: Module writing guide needs to be updated for new datatypes.

In particular, the executor example and datatypes in the module definition need to be updated.

Code layout

If you've followed the *basic project setup guide*, you'll have a project with a directory structure like this:

```
myproject/
    pipeline.conf
    pimlico/
        bin/
        lib/
        src/
        ...
```

(continues on next page)

(continued from previous page)

```
    src/
        python/
```

If you've not already created the `src/python` directory, do that now.

This is where your custom Python code will live. You can put all of your custom module types and datatypes in there and use them in the same way as you use the Pimlico core modules and datatypes.

Add this option to the `[pipeline]` section of your config file, so Pimlico knows where to find your code:

```
python_path=src/python
```

To follow the conventions used in Pimlico's codebase, we'll create the following package structure in `src/python`:

```
src/python/myproject/
    __init__.py
    modules/
        __init__.py
    datatypes/
        __init__.py
```

Write a module

A Pimlico module consists of a Python package with a special layout. Every module has a file `info.py`. This contains the definition of the module's metadata: its inputs, outputs, options, etc.

Most modules also have a file `execute.py`, which defines the routine that's called when it's run. You should take care when writing `info.py` not to import any non-standard Python libraries or have any time-consuming operations that get run when it gets imported.

`execute.py`, on the other hand, will only get imported when the module is to be run, after dependency checks.

For the example below, let's assume we're writing a module called `nmf` and create the following directory structure for it:

```
src/python/myproject/modules/
    __init__.py
    nmf/
        __init__.py
        info.py
        execute.py
```

Easy start

To help you get started, Pimlico provides a wizard in the *newmodule* command.

This will ask you a series of questions, guiding you through the most common tasks in creating a new module. At the end, it will generate a template to get you started with your module's code. You then just need to fill in the gaps and write the code for what the module actually does.

Read on to learn more about the structure of modules, including things not covered by the wizard.

Metadata

Module metadata (everything apart from what happens when it's actually run) is defined in `info.py` as a class called `ModuleInfo`.

Here's a sample basic `ModuleInfo`, which we'll step through. (It's based on the Scikit-learn `matrix_factorization` module.)

```
from pimlico.core.dependencies.python import PythonPackageOnPip
from pimlico.core.modules.base import BaseModuleInfo
from pimlico.datatypes.arrays import ScipySparseMatrix, NumpyArray


class ModuleInfo(BaseModuleInfo):
    module_type_name = "nmf"
    module_readable_name = "Sklearn non-negative matrix factorization"
    module_inputs = [("matrix", ScipySparseMatrix)]
    module_outputs = [("w", NumpyArray), ("h", NumpyArray)]
    module_options = {
        "components": {
            "help": "Number of components to use for hidden representation",
            "type": int,
            "default": 200,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + \
               [PythonPackageOnPip("sklearn", "Scikit-learn")]
```

The `ModuleInfo` should always be a subclass of *`BaseModuleInfo`*. There are some subclasses that you might want to use instead (e.g., see *Writing document map modules*), but here we just use the basic one.

Certain class-level attributes should pretty much always be overridden:

- `module_type_name`: A name used to identify the module internally
- `module_readable_name`: A human-readable short description of the module
- `module_inputs`: Most modules need to take input from another module (though not all)
- `module_outputs`: Describes the outputs that the module will produce, which may then be used as inputs to another module

Inputs are given as pairs `(name, type)`, where `name` is a short name to identify the input and `type` is the datatype that the input is expected to have. Here, and most commonly, this is a subclass of *`PimlicoDatatype`* and Pimlico will check that a dataset supplied for this input is either of this type, or has a type that is a subclass of this.

Here we take just a single input: a sparse matrix.

Outputs are given in a similar way. It is up to the module's executor (see below) to ensure that these outputs get written, but here we describe the datatypes that will be produced, so that we can use them as input to other modules.

Here we produce two Numpy arrays, the factorization of the input matrix.

Dependencies: Since we require Scikit-learn to execute this module, we override `get_software_dependencies()` to specify this. As Scikit-learn is available through Pip, this is very easy: all we need to do is specify the Pip package name. Pimlico will check that Scikit-learn is installed before executing the module and, if not, allow it to be installed automatically.

Finally, we also define some **options**. The values for these can be specified in the pipeline config file. When the `ModuleInfo` is instantiated, the processed options will be available in its `options` attribute. So, for ex-

ample, we can get the number of components (specified in the config file, or the default of 200) using `info.options["components"]`.

Executor

Here is a sample executor for the module info given above, placed in the file `execute.py`.

```
from pimlico.core.modules.base import BaseModuleExecutor
from pimlico.datatypes.arrays import NumpyArrayWriter
from sklearn.decomposition import NMF

class ModuleExecutor(BaseModuleExecutor):
    def execute(self):
        input_matrix = self.info.get_input("matrix").array
        self.log.info("Loaded input matrix: %s" % str(input_matrix.shape))

        # Convert input matrix to CSR
        input_matrix = input_matrix.tocsr()
        # Initialize the transformation
        components = self.info.options["components"]
        self.log.info("Initializing NMF with %d components" % components)
        nmf = NMF(components)

        # Apply transformation to the matrix
        self.log.info("Fitting NMF transformation on input matrix" % transform_type)
        transformed_matrix = transformer.fit_transform(input_matrix)

        self.log.info("Fitting complete: storing H and W matrices")
        # Use built-in Numpy array writers to output results in an appropriate format
        with NumpyArrayWriter(self.info.get_absolute_output_dir("w")) as w_writer:
            w_writer.set_array(transformed_matrix)
        with NumpyArrayWriter(self.info.get_absolute_output_dir("h")) as h_writer:
            h_writer.set_array(transformer.components_)
```

The executor is always defined as a class in `execute.py` called `ModuleExecutor`. It should always be a subclass of `BaseModuleExecutor` (though, again, note that there are more specific subclasses and class factories that we might want to use in other circumstances).

The `execute()` method defines what happens when the module is executed.

The instance of the module's `ModuleInfo`, complete with **options** from the pipeline config, is available as `self.info`. A standard Python **logger** is also available, as `self.log`, and should be used to keep the user updated on what's going on.

Getting hold of the **input data** is done through the module info's `get_input()` method. In the case of a Scipy matrix, here, it just provides us with the matrix as an attribute.

Then we do whatever our module is designed to do. At the end, we write the output data to the appropriate output directory. This should always be obtained using the `get_absolute_output_dir()` method of the module info, since Pimlico takes care of the exact location for you.

Most Pimlico datatypes provide a corresponding **writer**, ensuring that the output is written in the correct format for it to be read by the datatype's reader. When we leave the `with` block, in which we give the writer the data it needs, this output is written to disk.

Pipeline config

Our module is now ready to use and we can refer to it in a pipeline config file. We'll assume we've prepared a suitable Scipy sparse matrix earlier in the pipeline, available as the default output of a module called `matrix`. Then we can add section like this to use our new module:

```
[matrix]
...(Produces sparse matrix output)...

[factorize]
type=myproject.modules.nmf
components=300
input=matrix
```

Note that, since there's only one input, we don't need to give its name. If we had defined multiple inputs, we'd need to specify this one as `input_matrix=matrix`.

You can now run the module as part of your pipeline in the usual ways.

Skeleton new module

To make developing a new module a little quicker, here's a skeleton module info and executor.

```
from pimlico.core.modules.base import BaseModuleInfo

class ModuleInfo(BaseModuleInfo):
    module_type_name = "NAME"
    module_readable_name = "READABLE NAME"
    module_inputs = [("NAME", REQUIRED_TYPE)]
    module_outputs = [("NAME", PRODUCED_TYPE)]
    # Delete module_options if you don't need any
    module_options = {
        "OPTION_NAME": {
            "help": "DESCRIPTION",
            "type": TYPE,
            "default": VALUE,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + [
            # Add your own dependencies to this list
            # Remove this method if you don't need to add any
        ]
```

```
from pimlico.core.modules.base import BaseModuleExecutor

class ModuleExecutor(BaseModuleExecutor):
    def execute(self):
        input_data = self.info.get_input("NAME")
        self.log.info("MESSAGES")

        # DO STUFF

        with SOME_WRITER(self.info.get_absolute_output_dir("NAME")) as writer:
            # Do what the writer requires
```

1.1.6 Writing document map modules

Todo: Write a guide to building document map modules.

For now, the skeletons below are a useful starting point, but there should be a more fulsome explanation here of what document map modules are all about and how to use them.

Todo: Document map module guides needs to be updated for new datatypes.

Skeleton new module

To make developing a new module a little quicker, here's a skeleton module info and executor for a document map module. It follows the most common method for defining the executor, which is to use the multiprocessing-based executor factory.

```
from pimlico.core.modules.map import DocumentMapModuleInfo
from pimlico.datatypes.tar import TarredCorpusType

class ModuleInfo(DocumentMapModuleInfo):
    module_type_name = "NAME"
    module_readable_name = "READABLE NAME"
    module_inputs = [("NAME", TarredCorpusType(DOCUMENT_TYPE))]
    module_outputs = [("NAME", PRODUCED_TYPE)]
    module_options = {
        "OPTION_NAME": {
            "help": "DESCRIPTION",
            "type": TYPE,
            "default": VALUE,
        },
    }

    def get_software_dependencies(self):
        return super(ModuleInfo, self).get_software_dependencies() + [
            # Add your own dependencies to this list
        ]

    def get_writer(self, output_name, output_dir, append=False):
        if output_name == "NAME":
            # Instantiate a writer for this output, using the given output dir
            # and passing append in as a kwarg
            return WRITER_CLASS(output_dir, append=append)
```

A bare-bones executor:

```
from pimlico.core.modules.map.multiproc import multiprocessing_executor_factory


def process_document(worker, archive_name, doc_name, *data):
    # Do something to process the document...

    # Return an object to send to the writer
    return output
```

(continues on next page)

(continued from previous page)

```
ModuleExecutor = multiprocessing_executor_factory(process_document)
```

Or getting slightly more sophisticated:

```
from pimlico.core.modules.map.multiproc import multiprocessing_executor_factory


def process_document(worker, archive_name, doc_name, *data):
    # Do something to process the document

    # Return a tuple of objects to send to each writer
    # If you only defined a single output, you can just return a single object
    return output1, output2, ...


# You don't have to, but you can also define pre- and postprocessing
# both at the executor level and worker level

def preprocess(executor):
    pass


def postprocess(executor, error=None):
    pass


def set_up_worker(worker):
    pass


def tear_down_worker(worker, error=None):
    pass


ModuleExecutor = multiprocessing_executor_factory(
    process_document,
    preprocess_fn=preprocess, postprocess_fn=postprocess,
    worker_set_up_fn=set_up_worker, worker_tear_down_fn=tear_down_worker,
)
```

1.1.7 Filter modules

Filter modules appear in pipeline config, but never get executed directly, instead producing their output on the fly when it is needed.

There are two types of filter modules in Pimlico:

- All *document map modules* can be used as filters.
- Other modules may be defined in such a way that they always function as filters.

Using document map modules as filters

See *this guide* for how to create document map modules, which process each document in an input iterable corpus, producing one document in the output corpus for each. Many of the core Pimlico modules are document map modules.

Any document map module can be used as a filter simply by specifying `filter=True` in its options. It will then not appear in the module execution schedule (output by the `status` command), but will get executed on the fly by any module that uses its output. It will be initialized when the downstream module starts accessing the output, and then the single-document processing routine will be run on each document to produce the corresponding output document as the downstream module iterates over the corpus.

It is possible to chain together filter modules in sequence.

Other filter modules

Todo: Filter module guide needs to be updated for new datatypes. This section is currently completely wrong – **ignore it!** This is quite a substantial change.

The difficulty of describing what you need to do here suggests we might want to provide some utilities to make this easier!

A module can be defined so that it always functions as a filter by setting `module_executable=False` on its module-info class. Pimlico will assume that its outputs are ready as soon as its inputs are ready and will not try to execute it. The module developer must ensure that the outputs get produced when necessary.

This form of filter is typically appropriate for very simple transformations of data. For example, it might perform a simple conversion of one datatype into another to allow the output of a module to be used as if it had a different datatype. However, it is possible to do more sophisticated processing in a filter module, though the implementation is a little more tricky (`tar_filter` is an example of this).

Defining

Define a filter module something like this:

```
class ModuleInfo(BaseModuleInfo):
    module_type_name = "my_module_name"
    module_executable = False  # This is the crucial instruction to treat this as a␣
↪filter
    module_inputs = []         # Define inputs
    module_outputs = []        # Define at least one output, which we'll produce as␣
↪needed
    module_options = {}        # Any options you need

    def instantiate_output_datatype(self, output_name, output_datatype, **kwargs):
        # Here we produce the desired output datatype,
        # using the inputs acquired from self.get_input(name)
        return MyOutputDatatype()
```

You don't need to create an `execute.py`, since it's not executable, so Pimlico will not try to load a module executor. Any processing you need to do should be put inside the datatype, so that it's performed when the datatype is used (e.g. when iterating over it), but not when `instatiate_output_datatype()` is called or when the datatype is instantiated, as these happen every time the pipeline is loaded.

A trick that can be useful to wrap up functionality in a filter datatype is to define a new datatype that does the necessary processing on the fly and to set its class attribute `emulated_datatype` to point to a datatype class that should be used instead for the purposes of type checking. The built-in `tar_filter` module uses this trick.

Either way, you should **take care with imports**. Remember that the `execute.py` of executable modules is only imported when a module is to be run, meaning that we can load the pipeline config without importing any dependencies

needed to run the module. If you put processing in a specially defined datatype class that has dependencies, make sure that they're not imported at the top of `info.py`, but only when the datatype is used.

1.1.8 Multistage modules

Multistage modules are used to encapsulate a module than is executed in several consecutive runs. You can think of each stage as being its own module, but where the whole sequence of modules is always executed together. The multistage module simply chains together these individual modules so that you only include a single module instance in your pipeline definition.

One common example of a use case for multistage modules is where some fairly time-consuming preprocessing needs to be done on an input dataset. If you put all of the processing into a single module, you can end up in an irritating situation where the lengthy data preprocessing succeeds, but something goes wrong in the main execution code. You then fix the problem and have to run all the preprocessing again.

Most obvious solution to this is to separate the preprocessing and main execution into two separate modules. But then, if you want to reuse you module sometime in the future, you have to remember to always put the preprocessing module before the main one in your pipeline (or infer this from the datatypes!). And if you have more than these two modules (say, a sequence of several, or preprocessing of several inputs) this starts to make pipeline development frustrating.

A multistage module groups these internal modules into one logical unit, allowing them to be used together by including a single module instance and also to share parameters.

Defining a multistage module

Component stages

The first step in defining a multistage module is to define its individual stages. These are actually defined in exactly the same way as normal modules. (This means that they can also be used separately.)

If you're writing these modules specifically to provide the stages of your multistage module (rather than tying together already existing modules for convenience), you probably want to put them all in subpackages.

For an ordinary module, *we used the directory structure*:

```
src/python/myproject/modules/
    __init__.py
    mymodule/
        __init__.py
        info.py
        execute.py
```

Now, we'll use something like this:

```
src/python/myproject/modules/
    __init__.py
    my_ms_module/
        __init__.py
        info.py
        module1/
            __init__.py
            info.py
            execute.py
        module2/
            __init__.py
```

(continues on next page)

(continued from previous page)

```
            info.py
            execute.py
```

Note that `module1` and `module2` both have the typical structure of a module definition: an `info.py` to define the module-info, and an `execute.py` to define the executor. At the top level, we've just got an `info.py`. It's in here that we'll define the multistage module. We don't need an `execute.py` for that, since it just ties together the other modules, using their executors at execution time.

Multistage module-info

With our component modules that constitute the stages defined, we now just need to tie them together. We do this by defining a module-info for the multistage module in its `info.py`. Instead of subclassing `BaseModuleInfo`, as usual, we create the `ModuleInfo` class using the factory function *`multistage_module()`*.

```
ModuleInfo = multistage_module("module_name",
    [
        # Stages to be defined here...
    ]
)
```

In other respects, this module-info works in the same way as usual: it's a class (return by the factory) called `ModuleInfo` in the `info.py`.

`multistage_module()` takes two arguments: a module name (equivalent to the *module_name* attribute of a normal module-info) and a list of instances of *`ModuleStage`*.

Connecting inputs and outputs

Connections between the outputs and inputs of the stages work in a very similar way to connections between module instances in a pipeline. The same type checking system is employed and data is passed between the stages (i.e. between consecutive executions) as if the stages were separate modules.

Each stage is defined as an instance of *`ModuleStage`*:

```
[
    ModuleStage("stage_name", TheModuleInfoClass, connections=[...], output_
↪connections=[...])
]
```

The parameter `connections` defines how the stage's inputs are connected up to either the outputs of previous stages or inputs to the multistage module. Just like in pipeline config files, if no explicit input connections are given, the default input to a stage is connected to the default output from the previous one in the list.

There are two classes you can use to define input connections.

`InternalModuleConnection` This makes an explicit connection to the output of another stage.

> You must specify the name of the input (to this stage) that you're connecting. You may specify the name of the output to connect it to (defaults to the default output). You may also give the name of the stage that the output comes from (defaults to the previous one).

```
[
    ModuleStage("stage1", FirstInfo),
    # FirstInfo has an output called "corpus", which we connect explicitly to the␣
↪next stage
```

(continues on next page)

(continued from previous page)

```
    # We could leave out the "corpus" here, if it's the default output from␣
↪FirstInfo
    ModuleStage("stage2", SecondInfo, connections=[InternalModuleConnection("data
↪", "corpus")]),
    # We connect the same output from stage1 to stage3
    ModuleStage("stage3", ThirdInfo, connections=[InternalModuleConnection("data",
↪ "corpus", "stage1")]),
]
```

***ModuleInputConnection*:** This makes a connection to an input to the whole multistage module.

Note that you don't have to explicitly define the multistage module's inputs anywhere: you just mark certain inputs to certain stages as coming from outside the multistage module, using this class.

```
[
    ModuleStage("stage1", FirstInfo,  [ModuleInputConnection("raw_data")]),
    ModuleStage("stage2", SecondInfo, [InternalModuleConnection("data", "corpus
↪")]),
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1")]),
]
```

Here, the module type `FirstInfo` has an input called `raw_data`. We've specified that this needs to come in directly as an input to the multistage module – when we use the multistage module in a pipeline, it must be connected up with some earlier module.

The multistage module's input created by doing this will also have the name `raw_data` (specified using a parameter `input_raw_data` in the config file). You can override this, if you want to use a different name:

```
[
    ModuleStage("stage1", FirstInfo,  [ModuleInputConnection("raw_data", "data
↪")]),
    ModuleStage("stage2", SecondInfo, [InternalModuleConnection("data", "corpus
↪")]),
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1")]),
]
```

This would be necessary if two stages both had inputs called `raw_data`, which you want to come from different data sources. You would then simply connect them to different inputs to the multistage module:

```
[
    ModuleStage("stage1", FirstInfo,  [ModuleInputConnection("raw_data", "first_
↪data")]),
    ModuleStage("stage2", SecondInfo, [ModuleInputConnection("raw_data", "second_
↪data")]),
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1")]),
]
```

Conversely, you might deliberately connect the inputs from two stages to the same input to the multistage module, by using the same multistage input name twice. (Of course, the two stages are not required to have overlapping input names for this to work.) This will result in the multistage just requiring one input, which get used by both stages.

```
[
    ModuleStage("stage1", FirstInfo,
```

(continues on next page)

(continued from previous page)

```
                [ModuleInputConnection("raw_data", "first_data"),␣
↪ModuleInputConnection("dict", "vocab")]),
    ModuleStage("stage2", SecondInfo,
                [ModuleInputConnection("raw_data", "second_data"),␣
↪ModuleInputConnection("vocabulary", "vocab")]),
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1")]),
]
```

By default, the multistage module has just a single output: the default output of the last stage in the list. You can specify any of the outputs of any of the stages to be provided as an output to the multistage module. Use the `output_connections` parameter when defining the stage.

This parameter should be a list of instances of *ModuleOutputConnection*. Just like with input connections, if you don't specify otherwise, the multistage module's output will have the same name as the output from the stage module. But you can override this when giving the output connection.

```
[
    ModuleStage("stage1", FirstInfo, [ModuleInputConnection("raw_data", "first_data
↪")]),
    ModuleStage("stage2", SecondInfo, [ModuleInputConnection("raw_data", "second_data
↪")],
                output_connections=[ModuleOutputConnection("model")]),    # This␣
↪output will just be called "model"
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1"),
                output_connections=[ModuleOutputConnection("model", "stage3_model")]),
]
```

Module options

The parameters of the multistage module that can be specified when it is used in a pipeline config (those usually defined in the `module_options` attribute) include all of the options to all of the stages. The option names are simply `<stage_name>_<option_name>`.

So, in the above example, if `FirstInfo` has an option called `threshold`, the multistage module will have an option `stage1_threshold`, which gets passed through to `stage1` when it is run.

Often you might wish to specify one parameter to the multistage module that gets used by several stages. Say `stage2` had a `cutoff` parameter and we always wanted to use the same value as the `threshold` for `stage1`. Instead of having to specify `stage1_threshold` and `stage2_cutoff` every time in your config file, you can assign a single name to an option (say `threshold`) for the multistage module, whose value gets passed through to the appropriate options of the stages.

Do this by specifying a dictionary as the `option_connections` parameter to *ModuleStage*, whose keys are names of the stage module type's options and whose values are the new option names for the multistage module that you want to map to those stage options. You can use the same multistage module option name multiple times, which will cause only a single option to be added to the multistage module (using the definition from the first stage), which gets mapped to multiple stage options.

To implement that above example, you would give:

```
[
    ModuleStage("stage1", FirstInfo, [ModuleInputConnection("raw_data", "first_data
↪")],
```

(continues on next page)

(continued from previous page)

```
                option_connections={"threshold": "threshold"}),
    ModuleStage("stage2", SecondInfo, [ModuleInputConnection("raw_data", "second_data
↪")],
                [ModuleOutputConnection("model")],
                option_connections={"cutoff": "threshold"}),
    ModuleStage("stage3", ThirdInfo,  [InternalModuleConnection("data", "corpus",
↪"stage1"),
                [ModuleOutputConnection("model", "stage3_model")]),
]
```

If you know that the different stages have distinct option name, or that they should always tie their values together where their option names overlap, you can set `use_stage_option_names=True` on the stages. This will cause the stage-name prefix not to be added to the option name when connecting it to the multistage module's option.

You can also force this behaviour for all stages by setting `use_stage_option_names=True` when you call *`multistage_module()`*. Any explicit option name mappings you provide via `option_connections` will override this.

Running

To run a multistage module once you've used it in your pipeline config, you run one stage at a time, as if they were separate module instances.

Say we've used the above multistage module in a pipeline like so:

```
[model_train]
type=myproject.modules.my_ms_module
stage1_threshold=10
stage2_cutoff=10
```

The normal way to run this module would be to use the `run` command with the module name:

```
./pimlico.sh mypipeline.conf run model_train
```

If we do this, Pimlico will choose the next unexecuted stage that's ready to run (presumably `stage1` at this point). Once that's done, you can run the same command again to execute `stage2`.

You can also select a specific stage to execute by using the module name `<ms_module_name>:<stage_name>`, e.g. `model_train:stage2`. (Note that `stage2` doesn't actually depend on `stage1`, so it's perfectly plausible that we might want to execute them in a different order.)

If you want to execute multiple stages at once, just use this scheme to specify each of them as a module name for the run command. Remember, Pimlico can take any number of modules and execute them in sequence:

```
./pimlico.sh mypipeline.conf run model_train:stage1 model_train:stage2
```

Or, if you want to execute all of them, you can use the stage name `*` or `all` as a shorthand:

```
./pimlico.sh mypipeline.conf run model_train:all
```

Finally, if you're not sure what stages a multistage module has, use the module name `<ms_module_name>:?`. The run command will then just output a list of stages and exit.

1.1.9 Running on multiple computers

Multiple servers

In most of the examples, we've been setting up a pipeline, with a config file, some source code and some data, all on one machine. Then we run each module in turn, checking that it has all the software and data that it needs to run.

But it's not unusual to find yourself needing to process a dataset across different computers. For example, you have access to a server with lots of CPUs and one module in your pipeline would benefit greatly from parallelizing lots of little tasks over them. However, you don't have permission to install software on that server that you need for another module.

This is not a problem: you can simply put your config file and code on both machines. After running one module on one machine, you copy over its output to the place on the other machine where Pimlico expects to find it. Then you're ready to run the next module on the second machine.

Pimlico is designed to handle this situation nicely.

- **It doesn't expect software requirements for all modules to be satisfied before you can run any of them.** Software dependencies are checked only for modules about to be run and the code used to execute a module is not even loaded until you actually run the module.
- **It doesn't require you to execute your pipeline in order.** If the output from a module is available where it's expected to be, you can happily run any modules that take that data as input, even if the pipeline up to that point doesn't appear to have been executed (e.g. if it's been run on another machine).
- **It provides you with tools to make it easier to copy data between machines.** You can easily copy the output data from one module to the appropriate location on another server, so it's ready to be used as input to another module there.

Copying data between computers

Let's assume you've got your pipeline set up, with identical config files, on two computers: `server_a` and `server_b`. You've run the first module in your pipeline, `module1`, on `server_a` and want to run the next, `module2`, which takes input from `module1`, on `server_b`.

The procedure is as follows:

- **Dump** the data from the pipeline on `server_a`. This packages up the output data for a module in a single file.
- **Copy** the dumped file from `server_a` to `server_b`, in whatever way is most convenient, e.g., using `scp`.
- **Load** the dumped file into the pipeline on `server_b`. This unpacks the data directory for the file and puts it in Pimlico's data directory for the module.

For example, on `server_a`:

```
$ ./pimlico.sh pipeline.conf dump module1
$ scp ~/module1.tar.gz server_b:~/
```

Note that the `dump` command created a `.tar.gz` file in your home directory. If you want to put it somewhere else, use the `--output` option to specify a directory. The file is named after the module that you're dumping.

Now, log into `server_b` and load the data.

```
$ ./pimlico.sh pipeline.conf load ~/module1.tar.gz
```

Now `module1`'s output data is in the right place and ready for use by `module2`.

The `dump` and `load` commands can also process data for multiple modules at once. For example:

```
$ mkdir ~/modules
$ ./pimlico.sh pipeline.conf dump module1 ... module10 --output ~/modules
$ scp -r ~/modules server_b:~/
```

Then on `server_b`:

```
$ ./pimlico.sh pipeline.conf load ~/modules/*
```

Other issues

Aside from getting data between the servers, there are certain issues that often arise when running a pipeline across multiple servers.

- **Shared Pimlico codebase. If you share the directory that contains Pimlico's code across servers** (e.g. NFS or `rsync`), you can have problems resulting from sharing the libraries it installs. See *instructions for using multiple virtualenvs* for the solution.
- **Shared home directory. If you share your home directory across servers, using the same `.pimlico` local** config file might be a problem. See *Local configuration* for various possible solutions.

1.1.10 Documenting your own code

Pimlico's documentation is produced using Sphinx. The Pimlico codebase includes a tool for generating documentation of Pimlico's built-in modules, including things like a table of the module's available config options and its input and outputs.

You can also use this tool yourself to generate documentation of your own code that uses Pimlico. Typically, you will use in your own project some of Pimlico's built-in modules and some of your own.

Refer to Sphinx's documentation for how to build normal Sphinx documentation – writing your own ReST documents and using the apidoc tool to generate API docs. Here we describe how to create a basic Sphinx setup that will generate a reference for your custom Pimlico modules.

It is assumed that you've got a working Pimlico setup and have already successfully written some modules.

Basic doc setup

Create a `docs` directory in your project root (the directory in which you have `pimlico/` and your own `src/`, etc).

Put a Sphinx `conf.py` in there. You can start from the very basic skeleton `here`.

You'll also want a `Makefile` to build your docs with. You can use the basic Sphinx one as a starting point. `Here's` a version of that that already includes an extra target for building your module docs.

Finally, create a root document for your documentation, `index.rst`. This should include a table of contents which includes the generated module docs. You can use `this one` as a template.

Building the module docs

Take a look in the `Makefile` (if you've used our one as a starting point) and set the variables at the top to point to the Python package that contains the Pimlico modules you want to document.

The make target there runs the tool `modulegen` in the Pimlico codebase. Just run, in the `docs/`:

```
make modules
```

You can also do this manually:

```
python -m pimlico.utils.docs.modulegen --path python.path.to.modules modules/
```

(The Pimlico codebase must, of course, be importable. The simplest way to ensure this is to use Pimlico's `python` alias in its `bin/` directory.)

There is now a set of `.rst` files in the `modules/` output directory, which can be built using Sphinx by running `make html`.

Your beautiful docs are now in the `_build/` directory!

1.2 Core docs

A set of articles on the core aspects and features of Pimlico.

1.2.1 Downloading Pimlico

To start a new project using Pimlico, download the newproject.py script. It will create a template pipeline config file to get you started and download the latest version of Pimlico to accompany it.

See *Setting up a new project using Pimlico* for more detail.

Pimlico's source code is available on on Github.

Manual setup

If for some reason you don't want to use the `newproject.py` script, you can set up a project yourself. Download Pimlico from Github.

Simply download the whole source code as a `.zip` or `.tar.gz` file and uncompress it. This will produce a directory called `pimlico`, followed by a long incomprehensible string, which you can rename simply `pimlico`.

Pimlico has a few basic dependencies, but these will be automatically downloaded the first time you load it.

1.2.2 Pipeline config

A Pimlico pipeline, as read from a config file (*`pimlico.core.config.PipelineConfig`*) contains all the information about the pipeline being processed and provides access to specific modules in it. A config file looks much like a standard `.ini` file, with sections headed by `[section_name]` headings, containing key-value parameters of the form `key=value`.

Each section, except for `vars` and `pipeline`, defines a module instance in the pipeline. Some of these can be executed, others act as filters on the outputs of other modules, or input readers.

Module instances

The main components of a pipeline config file are sections defining module instances. They load a module of a particular type, giving a set of options controlling what it does and specifying inputs, connecting it up to the outputs from previous modules.

Each section that defines a module has a `type` parameter. Usually, this is a fully-qualified Python package name that leads to the module type's Python code (that package containing the `info` Python module).

A typical module instance section looks like this:

```
[mymodule]
type=pimlico.modules.corpora.subset
input_corpus=corpus_module.some_output
size=100
```

Here the *subset* module is instantiated, with the option `size=100`, specifying how big a subset to create. This module takes a single input, called `corpus`, which is here connected to the output of a previous module instance. Inputs are connected using parameters of the form `input_<input-name>`. They give the name of a previous module instance and, optionally, the name of the output of that module that we will get input from. (If not specified, the module's default output will be used.)

The documentation for each module gives details of:

- what options it can take, including their types and default values
- what inputs it takes and their datatypes
- what outputs it produces and their datatypes.

A large set of *module types* for different tasks, ranging from simple dataset manipulation to advanced natual language processing or machine learning tasks, comes built into Pimlico and each is *fully documented*.

Many of the core modules are associated with test pipelines, which serve both as unit tests for the module and examples of how it can be used.

A special type of module is `alias`. This simply defines a module alias – an alternative name for an already defined module. It should have exactly one other parameter, `input`, specifying the name of the module we're aliasing.

Special sections

- **vars:** May contain any variable definitions, to be used later on in the pipeline. Further down, expressions like `%(varname)s` will be expanded into the value assigned to `varname` in the vars section.
- **pipeline:** Main pipeline-wide configuration. The following options are required for every pipeline:
  - `name`: a single-word name for the pipeline, used to determine where files are stored
  - `release`: the release of Pimlico for which the config file was written. It is considered compatible with later minor versions of the same major release, but not with later major releases. Typically, a user receiving the pipeline config will get hold of an appropriate version of the Pimlico codebase to run it with.

  Other optional settings:

  - `python_path`: a path or paths, relative to the directory containing the config file, in which Python modules/packages used by the pipeline can be found. Typically, a config file is distributed with a directory of Python code providing extra modules, datatypes, etc. Multiple paths are separated by colons (:).

Special variable substitutions

Certain variable substitutions are always available, in addition to those defined in `vars` sections. Use them anywhere in your config file with an expression like `%(varname)s` (note the `s` at the end).

- **`pimlico_root`:** Root directory of Pimlico, usually the directory `pimlico/` within the project directory.

- **project_root:** Root directory of the whole project. Current assumed to always be the parent directory of `pimlico_root`.
- **output_dir:** Path to output dir (usually `output` in Pimlico root).
- **home:** Running user's home directory (on Unix and Windows, see Python's `os.path.expanduser()`).
- **test_data_dir:** Directory in Pimlico distribution where test data is stored (`test/data` in Pimlico root). Used in test pipelines, which take all their input data from this directory.

For example, to point a parameter to a file located within the project root:

```
param=%(project_root)s/data/myfile.txt
```

Directives

Certain special directives are processed when reading config files. They are lines that begin with `%%`, followed by the directive name and any arguments.

- **variant:** Allows a line to be included only when loading a particular variant of a pipeline. For more detail on pipeline variants, see *Pipeline variants*.

  The variant name is specified as part of the directive in the form: `variant:variant_name`. You may include the line in more than one variant by specifying multiple names, separated by commas (and no spaces). You can use the default variant "main", so that the line will be left out of other variants. The rest of the line, after the directive and variant name(s) is the content that will be included in those variants.

  ```
  [my_module]
  type=path.to.module
  %%variant:main size=52
  %%variant:smaller size=7
  ```

  An alternative notation for the variant directive is provided to make config files more readable. Instead of `variant:variant_name`, you can write `(variant_name)`. So the above example becomes:

  ```
  [my_module]
  type=path.to.module
  %%(main) size=52
  %%(smaller) size=7
  ```

- **novariant:** A line to be included only when not loading a variant of the pipeline. Equivalent to `variant:main`.

  ```
  [my_module]
  type=path.to.module
  %%novariant size=52
  %%variant:smaller size=7
  ```

- **include:** Include the entire contents of another file. The filename, specified relative to the config file in which the directive is found, is given after a space.
- **abstract:** Marks a config file as being abstract. This means that Pimlico will not allow it to be loaded as a top-level config file, but only allow it to be included in another config file.
- **copy:** Copies all config settings from another module, whose name is given as the sole argument. May be used multiple times in the same module and later copies will override earlier. Settings given explicitly in the module's config override any copied settings.

All parameters are copied, including things like `type`. Any parameter can be overridden in the copying module instance. Any parameter can be excluded from the copy by naming it after the module name. Separate multiple exclusions with spaces.

The directive even allows you to copy parameters from multiple modules by using the directive multiple times, though this is not very often useful. In this case, the values are copied (and overridden) in the order of the directives.

For example, to reuse all the parameters from `module1` in `module2`, only specifying them once:

```
[module1]
type=some.module.type
input=moduleA
param1=56
param2=never
param3=0.75

[module2]
# Copy all params from module1
%%copy module1
# Override the input module
input=moduleB
```

Multiple parameter values

Sometimes you want to write a whole load of modules that are almost identical, varying in just one or two parameters. You can give a parameter multiple values by writing them separated by vertical bars (`|`). The module definition will be expanded to produce a separate module for each value, with all the other parameters being identical.

For example, this will produce three module instances, all having the same `num_lines` parameter, but each with a different `num_chars`:

```
[my_module]
type=module.type.path
num_lines=10
num_chars=3|10|20
```

You can even do this with multiple parameters of the same module and the expanded modules will cover all combinations of the parameter assignments.

For example:

```
[my_module]
type=module.type.path
num_lines=10|50|100
num_chars=3|10|20
```

Tying alternatives

You can change the behaviour of alternative values using the `tie_alts` option. `tie_alts=T` will cause parameters within the same module that have multiple alternatives to be expanded in parallel, rather than taking the product of the alternative sets. So, if `option_a` has 5 values and `option_b` has 5 values, instead of producing 25 pipeline modules, we'll only produce 5, matching up each pair of values in their alternatives.

```
[my_module]
type=module.type.path
tie_alts=T
option_a=1|2|3|4|5
option_b=one|two|three|four|five
```

If you want to tie together the alternative values on some parameters, but not others, you can specify groups of parameter names to tie using the `tie_alts` option. Each group is separated by spaces and the names of parameters to tie within a group are separated by `|` s. Any parameters that have alternative values but are not specified in one of the groups are not tied to anything else.

For example, the following module config will tie together `option_a`'s alternatives with `option_b`'s, but produce all combinations of them with `option_c` 's alternatives, resulting in 3*2=6 versions of the module (`my_module[option_a=1~option_b=one~option_c=x]`, `my_module[option_a=1~option_b=one~option_c=y]`, `my_module[option_a=2~option_b=two~option_c=x]` etc).

```
[my_module]
type=module.type.path
tie_alts=option_a|option_b
option_a=1|2|3
option_b=one|two|three
option_c=x|y
```

Using this method, you must give the parameter names in `tie_alts` exactly as you specify them in the config. For example, although for a particular module you might be able to specify a certain input (the default) using the name `input` or a specific name like `input_data`, these will not be recognised as being the same parameter in the process of expanding out the combinations of alternatives.

Naming alternatives

Each module will be given a distinct name, based on the varied parameters. If just one is varied, the names will be of the form `module_name[param_value]`. If multiple parameters are varied at once, the names will be `module_name[param_name0=param_value0~param_name1=param_value1~...]`. So, the first example above will produce: `my_module[3]`, `my_module[10]` and `my_module[20]`. And the second will produce: `my_module[num_lines=10~num_chars=3]`, `my_module[num_lines=10~num_chars=10]`, etc.

You can also specify your own identifier for the alternative parameter values, instead of using the values themselves (say, for example, if it's a long file path). Specify it surrounded by curly braces at the start of the value in the alternatives list. For example:

```
   [my_module]
   type=module.type.path
   file_path={small}/home/me/data/corpus/small_version|{big}/home/me/data/corpus/big_
↪version
```

This will result in the modules `my_module[small]` and `my_module[big]`, instead of using the whole file path to distinguish them.

An alternative approach to naming the expanded alternatives can be selected using the `alt_naming` parameter. The default behaviour described above corresponds to `alt_naming=full`. If you choose `alt_naming=pos`, the alternative parameter settings (using names where available, as above) will be distinguished like positional arguments, without making explicit what parameter each value corresponds to. This can make for nice concise names in cases where it's clear what parameters the values refer to.

If you specify `alt_naming=full` explicitly, you can also give a further option `alt_naming=full(inputnames)`. This has the effect of removing the `input_` from the start of named inputs. This often makes for intuitive module names, but is not the default behaviour, since there's no guarantee that the input name (without the initial `input_`) does not clash with an option name.

Another possibility, which is occasionally appropriate, is `alt_naming=option(<name>)`, where `<name>` is the name of an option that has alternatives. In this case, the names of the alternatives for the whole module will be taken directly from the alternative names on that option only. (E.g. specified by `{name}` or inherited from a previous module, see below). You may specify multiple option names, separated by commas, and the corresponding alt names will be separated by `~`. If there's only one option with alternatives, this is equivalent to `alt_naming=pos`. If there are multiple, it might often lead to name clashes. The circumstance in which this is most commonly appropriate is where you use `tie_alts=T`, so it's sufficient to distinguish the alternatives by the name associated with just one option.

Expanding alternatives down the pipeline

If a module takes input from a module that has been expanded into multiple versions for alternative parameter values, it too will automatically get expanded, as if all the multiple versions of the previous module had been given as alternative values for the input parameter. For example, the following will result in 3 versions of `my_module` (`my_module[1]`, etc) and 3 corresponding versions of `my_next_module` (`my_next_module[1]`, etc):

```
[my_module]
type=module.type.path
option_a=1|2|3

[my_next_module]
type=another.module.type.path
input=my_module
```

Where possible, names given to the alternative parameter values in the first module will be carried through to the next.

Structure: headed sections

By default, a pipeline is ultimately just a list of modules. The `status` command will show a long list of all the modules. In some cases, this can get very long and difficult to navigate.

You can add structure to your pipeline by adding section headings. This is done by starting comments with multiple "#"s. In other words, the headings are part of the comments that come in between modules.

Headings follow Markdown-style formatting (only the *atx*-style, not *Setext*). Since our comments begin with `#``s, the first ``# does not denote a heading. Subsequet "#"s produce further nested levels of headings.

```
## Data pre-processing
# In this section, we will pre-process the data.
# Note that the line with a ## at the start is a heading.
# These subsequent ones are not - they are just comments.

### Pre-processing step 1
# This is a sub-section
[my_module]
type=module.type.path
```

When you run the `status` command now, you will see the section headings collapsed by default. You can expand individual sections, or everything, using the command's options.

Use `--expand-all` (`-xa`) to expand the full tree of section headings and show all modules. Use `--expand` (`-x`) with a section number (get this from the collapsed tree) to expand a given section. E.g. `./pimlico.sh myconf.conf status -x 2.3.1`. You can given multiple sections by repeating the `-x` option.

You can also expand a full subtree of a given section by ending the section number with a dot. `./pimlico.sh myconf.conf status -x 2.3.`.

Module variables: passing information through the pipeline

When a pipeline is read in, each module instance has a set of *module variables* associated with it. In your config file, you may specify assignments to the variables for a particular module. Each module inherits all of the variable assignments from modules that it receives its inputs from.

The main reason for having module variables it to be able to do things in later modules that depend on what path through the pipeline an input came from. Once you have defined the sequence of processing steps that pass module variables through the pipeline, apply mappings to them, etc, you can use them in the parameters passed into modules.

Basic assignment

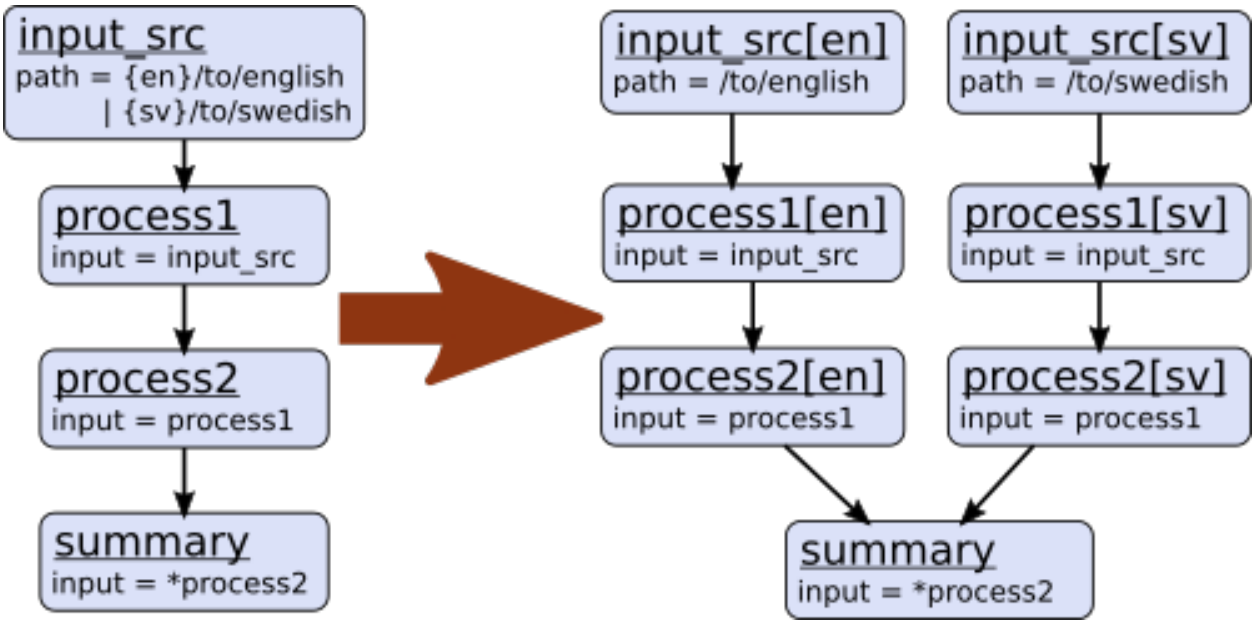
Module variables are set by including parameters in a module's config of the form `modvar_<name> = <value>`. This will assign `value` to the variable `name` for this module. The simplest form of assignment is just a string literal, enclosed in double quotes:

```
[my_module]
type=module.type.path
modvar_myvar = "Value of my variable"
```

Names of alternatives

Say we have a simple pipeline that has a single source of data, with different versions of the dataset for different languages (English and Swedish). A series of modules process each language in an identical way and, at the end, outputs from all languages are collected by a single `summary` module. This final module may need to know what language each of its incoming datasets represents, so that it can output something that we can understand.

The two languages are given as alternative values for a parameter `path`, and the whole pipeline gets automatically expanded into two paths for the two alternatives:



The `summary` module gets its two inputs for the two different languages as a multiple-input: this means we could expand this pipeline to as many languages as we want, just by adding to the `input_src` module's `path` parameter.

However, as far as `summary` is concerned, this is just a list of datasets – it doesn't know that one of them is English and one is Swedish. But let's say we want it to output a table of results. We're going to need some labels to identify the languages.

The solution is to add a module variable to the first module that takes different values when it gets expanded into two modules. For this, we can use the `altname` function in a modvar assignment: this assigns the name of the expanded module's alternative for a given parameter that has alternatives in the config.

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)
```

Now the expanded module `input_src[en]` will have the module variable `lang="en"` and the Swedish version `lang="sv"`. This value gets passed from module to module down the two paths in the pipeline.

Other assignment syntax

A further function `map` allows you to apply a mapping to a value, rather like a Python dictionary lookup. Its first argument is the value to be mapped (or anything that expands to a value, using modvar assignment syntax). The second is the mapping. This is simply a space-separated list of source-target mappings of the form `source -> target`. Typically both the sources and targets will be string literals.

Now we can give our languages legible names. (Here we're splitting the definition over multiple indented lines, as permitted by config file syntax, which makes the mapping easier to read.)

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=map(
    altname(path),
    "en" -> "English"
    "sv" -> "Svenska")
```

The assignments may also reference variable names, including those previously assigned to in the same module and those received from the input modules.

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)
modvar_lang_name=map(
    lang,
    "en" -> "English"
    "sv" -> "Svenska")
```

If a module gets two values for the same variable from multiple inputs, the first value will simply be overridden by the second. Sometimes it's useful to map module variables from specific inputs to different modvar names. For example, if we're combining two different languages, we might need to keep track of what the two languages we combined were. We can do this using the notation `input_name.var_name`, which refers to the value of module variable `var_name` that was received from input `input_name`.

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)

[combiner]
type=my.language.combiner
input_lang_a=lang_data
input_lang_b=lang_data
modvar_first_lang=lang_a.lang
modvar_second_lang=lang_b.lang
```

If a module inherits multiple values for the same variable from the **same input** (i.e. a multiple-input), they are all kept and treated as a list. The most common way to then use the values is via the `join` function. Like Python's `string.join`, this turns a list into a single string by joining the values with a given separator string. Use `join(sep, list)` to join the values coming from some list modvar `list` on the separator `sep`.

You can get the number of values in a list modvar using `len(list)`, which works just like Python's `len()`.

Use in module parameters

To make something in a module's execution dependent on its module variables, you can insert them into module parameters.

For example, say we want one of the module's parameters to make use of the `lang` variable we defined above:

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)
some_param=$(lang)
```

Note the difference to other variable substitutions, which use the `%(varname)s` notation. For modvars, we use the notation `$(varname)`.

We can also put the value in the middle of other text:

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)
some_param=myval-$(lang)-continues
```

The modvar processing to compute a particular module's set of variable assignments is performed before the substitution. This means that you can do any modvar processing specific to the module instance, in the various ways defined above, and use the resulting value in other parameters. For example:

```
[input_src]
path={en}/to/english | {sv}/to/swedish
modvar_lang=altname(path)
modvar_mapped_lang=map(lang,
      "en" -> "eng"
      "sv" -> "swe"
   )
some_param=$(mapped_lang)
```

You can also place in the `$(...)` construct any of the variable processing operations shown above for assignments to module variables. This is a little more concise than first assigning values to modvars, if you don't need to use the variables again anywhere else. For example:

```
[input_src]
path={en}/to/english | {sv}/to/swedish
some_param=$(map(altname(path)),
      "en" -> "eng"
      "sv" -> "swe"
   ))
```

Usage in module code

A module's executor can also retrieve the values assigned to module variables from the `module_variables` attribute of the module-info associated with the input dataset. Sometimes this can be useful when you are writing your own module code, though the above usage to pass values from (or dependent on) module variables into module parameters is more flexible, so should generally be preferred.

```
# Code in executor
# This is a MultipleInput-type input, so we get a list of datasets
datasets = self.info.get_input()
for d in datasets:
    language = d.module.module_variables["lang"]
```

1.2.3 Pipeline variants

You can create several different versions of a pipeline, called pipeline *variants* in a single config file. The data corresponding to each will be kept completely separate. This is useful when you want multiple versions of a pipeline that are almost identical, but have some small differences.

The most common use of this, though by no means the only, is to create a variant that is faster to run than the main pipeline for the purposes of quickly testing the whole pipeline during development.

Every pipeline has by default one variant, called `main`. You define other variants simply by using special directives to mark particular lines as belonging to a particular variant. Lines with no variant marking will appear in all variants.

Loading variants

If you don't specify otherwise when loading a pipeline, the `main` variant will be loaded. Use the `--variant` parameter (or `-v`) to specify another variant by name:

```
./pimlico.sh mypipeline.conf -v smaller status
```

To see a list of all available variants of a particular pipeline, use the *variants* command:

```
./pimlico.sh mypipeline.conf variants
```

Variant directives

Directives are processed when a pipeline config file is read in, before the file is parsed to build a pipeline. They are lines that begin with `%%`, followed by the directive name and any arguments. See *Directives* for details of other directives.

- **`variant:`** This line will be included only when loading a particular variant of a pipeline.

  The variant name is specified in the form: `variant:variant_name`. You may include the line in more than one variant by specifying multiple names, separated by commas (and no spaces). You can use the default variant "main", so that the line will be left out of other variants. The rest of the line, after the directive and variant name(s) is the content that will be included in those variants.

  ```
  [my_module]
  type=path.to.module
  %%variant:main size=52
  %%variant:smaller size=7
  ```

  An alternative notation makes config files more readable. Instead of `%%variant:variant_name`, write `%%(variant_name)`. So the above example becomes:

  ```
  [my_module]
  type=path.to.module
  %%(main) size=52
  %%(smaller) size=7
  ```

- **`novariant:`** A line to be included only when not loading a variant of the pipeline. Equivalent to `variant:main`.

  ```
  [my_module]
  type=path.to.module
  %%novariant size=52
  %%variant:smaller size=7
  ```

Example

The following example config file, defines one variant, `small`, aside from the default `main` variant.

```
[pipeline]
name=myvariants
release=0.8
python_path=%(project_root)s/src/python

# Load a dataset
[input_data]
type=pimlico.modules.input.text.raw_text_files
files=%(home)s/data/*
```

(continues on next page)

(continued from previous page)

```
# For the small version, we cut down the dataset to just 10 documents
# We don't need this module at all in the main variant
%%(small) [small_data]
%%(small) type=pimlico.modules.corpora.subset
%%(small) size=10

# Tokenize the text
# Control where the input data comes from in the different variants
# The main variant simply uses the full, uncut corpus
[tokenize]
type=pimlico.modules.text.simple_tokenize
%%(small) input=small_data
%%(main) input=input_data
```

The main variant will be loaded if you don't specify otherwise. In this version the module `small_data` doesn't exist at all and `tokenize` takes its input from `input_data`.

```
./pimlico.sh myvariants.conf status
```

You can load the small variant by giving its name on the command line. This includes the `small_data` module and `tokenize` gets its input from there, making it much faster to test.

```
./pimlico.sh myvariants.conf -v small status
```

1.2.4 Pimlico module structure

This document describes the code structure for Pimlico module types in full.

For a basic guide to writing your own modules, see *Writing Pimlico module types*.

Todo: Finish the missing parts of this doc below

For many generic or common tasks, you can use one of Pimlico's *built-in module types*, but often you will want to write your own module code to do some of the processing in your pipeline.

The *module-writing guide* takes you through how to write your own module type and use it in your pipeline. This document is a more comprehesive documentation of the structure of module definitions and execution code.

Code layout

A module is defined by a Python package containing at least two Python files (i.e. Python modules):

- `info.py`. This contains a class called `ModuleInfo` that provides all the structural information about a module: its inputs, outputs, options, etc. This is instantiated for each module of this type in a pipeline when the pipeline is loaded.
- `execute.py`. This contains a class called `ModuleExecutor` that has a method that is called when the module is run. The `execute` Python module is only imported when a module is about to be run, so is free to contain package imports that depend on having special software packages installed.

You should take care when writing `info.py` not to import any non-standard Python libraries or have any time-consuming operations that get run when it gets imported, as it's loaded whenever a pipeline containing that module type is loaded up, when checking module status for example, before software dependencies are resolved.

`execute.py`, on the other hand, will only get imported when the module is to be run, after dependency checks.

Pimlico provides a wizard in the *newmodule* command that guides you through the most common tasks in creating a new module. At the end, it will generate a template to get you started with your module's code. You then just need to fill in the gaps and write the code for what the module actually does. It does not, however, cover every type of module definition that you might want – just the most common cases.

Metadata: info.py

The ModuleInfo

Module metadata (everything apart from what happens when it's actually run) is defined in `info.py` as a class called `ModuleInfo`. Let's take the built-in module *corpus_stats* – which counts up tokens, sentences, etc in a corpus – as an example. (It's slightly modified here for the example.)

```
from pimlico.core.modules.base import BaseModuleInfo
from pimlico.datatypes import GroupedCorpus, NamedFile
from pimlico.datatypes.corpora.tokenized import TokenizedDocumentType


class ModuleInfo(BaseModuleInfo):
    module_type_name = "corpus_stats"
    module_readable_name = "Corpus statistics"
    module_inputs = [("corpus", GroupedCorpus(TokenizedDocumentType()))]
    module_outputs = [("stats", NamedFile("stats.json"))]
    module_options = {
        "min_sent_len": {
            "type": int,
            "help": "Filter out any sentences shorter than this length",
        }
    }
```

The `ModuleInfo` should always be a subclass of *BaseModuleInfo*. There are some subclasses that you might want to use instead (e.g., see *Writing document map modules*), but here we just use the basic one.

Certain class-level attributes should pretty much always be overridden:

- `module_type_name`: A name used to identify the module internally
- `module_readable_name`: A human-readable short description of the module
- `module_inputs`: Most modules need to take input from another module (though not all)
- `module_outputs`: Describes the outputs that the module will produce, which may then be used as inputs to another module

Inputs

Inputs are given as pairs `(name, type)`, where `name` is a short name to identify the input and `type` is the datatype that the input is expected to have. Here, and most commonly, this is an instance of a subclass of *PimlicoDatatype*. Pimlico will check that any dataset supplied for this input is of a compatible datatype.

Here we take just a single input. It is a corpus of the standard type that Pimlico uses for sequential document corpora `GroupedCorpus`. More specifically, it is a corpus with a document type of *TokenizedDocumentType*, or some sub-type.

Outputs

Outputs are given in a similar way. It is up to the module's executor (see below) to ensure that these outputs get written, but the ModuleInfo describes the datatypes that will be produced, so that we can use them as input to other modules.

In the example, we produce a single file containing the output of the analysis.

Once a module has been instantiated, its output names and types are available in its `available_outputs` attribute, which can be consulted by its executor and which is used for typechecking connections to later modules and loading the output datasets produced by the module.

Output groups

A module's outputs have no structure: each module just has a list of outputs identified by their names. The don't typically even have any particular order.

However, sometimes it can be useful to group together some of the outputs, so that they can easily be used later collectively. Say, for example, a module produces three corpora, each as a separate output, and also a `NamedFile` output containing some analysis. It is useful to be able to refer to the corpora as a group, rather than having to list them each by name, if for instance you are using all three to feed into a multiple-input to a later module. This becomes particularly important if the number of output corpora is not even statically defined: see below for how the number of outputs might depend on inputs and options.

A module can define named groups of outputs. Every module, by default, has a single module group, called `"all"`.

Once a module info has been instantiated, it has an attribute `output_groups` listing the groups. Each group is specified as `(group_name, [output_name1, ...])`.

In a config file, an output group name can be used in the same way as a single output name to specify where inputs to a module will come from: `module_name.output_group_name`. If a group name is given, instead of a single output name, it will be expanded into a comma-separated **list of output names** corresponding to that group. Of course, this will only work if the input in question is a **multiple-input**, allowing it to accept a comma-separated list of datasets as input.

Alternatively, you may use an output group to provide **alternative** datasets for an input, just as you usually would using `|``s. If you use ``altgroup(module_name.output_group_name)` as an input to a module, it will be expanded to `module_name.output_name1|module_name.output_name2|...` to provide each output in the group as an alternative input. (See *Pipeline config* for more on alternative inputs and parameters.)

Output groups are defined by the class attribute `module_output_groups` on the module info class and may be extended by overriding `build_output_groups()` to add more output groups containing further outputs added dependent on options and inputs.

Optional outputs

Todo: Document optional outputs.

Should include choose_optional_outputs_from_options(options, inputs) for deciding what optional outputs to include.

Outputs dependent on options

A module info can supply output names/types that are dependent on the module instance's inputs and options. This is done by overriding the method `provide_further_outputs()`. It is called once the ModuleInfo instance's `inputs` and `options` attributes have already been set and preprocessed.

It returns a list just like the statically defined `module_outputs` attribute: pairs of `(output_name, datatype_instance)`. Once the module info has been instantiated for a particular module in a pipeline, these outputs will be available in the `available_outputs` attribute, just like any that were defined statically.

If you override `provide_further_outputs()`, you should also give it a docstring describing the further outputs that may be added, how they are named and under what conditions they are added. This string will be included in the generated documentation for the module, underneath the table of outputs.

Options

Most modules define some **options** that provided control over exactly what the module does when executed. The values for these can be specified in the pipeline config file. When the `ModuleInfo` is instantiated, the processed options will be available in its `options` attribute.

In the example, there is one option that can be specified in the config file, like this:

```
[mymod]
type=pimlico.modules.corpora.corpus_stats
input=some_previous_mod
min_sent_len=5
```

The option definition provides some help text explaining what the option does, which is included in the module's documentation, which can be automatically produced using Sphinx (see *Documenting your own code*).

Its value can be accessed from within the executor's `execute()` method using: `self.info.options["min_sent_len"]`. By this point, the value from the config file has been checked and preprocessed, so it is an int.

Todo: Fully document module options, including: required, type checking/processing and other fancy features.

Software dependencies

Many modules rely on external Python packages or other software for their execution. The ModuleInfo specifies exactly what software is required in such a way that Pimlico can:

- check whether the software is available and runable;
- if possible, install the software if it's not available (e.g. Python packages installable via Pip);
- otherwise, provide instructions on how to install it;
- in some special cases, run initalization or other preparatory routines before the external software is loaded/run.

Todo: Further document specification of software dependencies

---

More extensive documentation of the Pimlico dependency system is provided in *Module dependencies*.

Execution: execute.py

Todo: This section is copied from *Pimlico module structure*. It needs to be re-written to provide more technical and comprehensive documentation of module execution.

Here is a sample executor for the module info given above, placed in the file `execute.py`.

```
from pimlico.core.modules.base import BaseModuleExecutor
from pimlico.datatypes.arrays import NumpyArrayWriter
from sklearn.decomposition import NMF

class ModuleExecutor(BaseModuleExecutor):
    def execute(self):
        input_matrix = self.info.get_input("matrix").array
        self.log.info("Loaded input matrix: %s" % str(input_matrix.shape))

        # Convert input matrix to CSR
        input_matrix = input_matrix.tocsr()
        # Initialize the transformation
        components = self.info.options["components"]
        self.log.info("Initializing NMF with %d components" % components)
        nmf = NMF(components)

        # Apply transformation to the matrix
        self.log.info("Fitting NMF transformation on input matrix" % transform_type)
        transformed_matrix = transformer.fit_transform(input_matrix)

        self.log.info("Fitting complete: storing H and W matrices")
        # Use built-in Numpy array writers to output results in an appropriate format
        with NumpyArrayWriter(self.info.get_absolute_output_dir("w")) as w_writer:
            w_writer.set_array(transformed_matrix)
        with NumpyArrayWriter(self.info.get_absolute_output_dir("h")) as h_writer:
            h_writer.set_array(transformer.components_)
```

The executor is always defined as a class in `execute.py` called `ModuleExecutor`. It should always be a subclass of `BaseModuleExecutor` (though, again, note that there are more specific subclasses and class factories that we might want to use in other circumstances).

The `execute()` method defines what happens when the module is executed.

The instance of the module's `ModuleInfo`, complete with **options** from the pipeline config, is available as `self.info`. A standard Python **logger** is also available, as `self.log`, and should be used to keep the user updated on what's going on.

Getting hold of the **input data** is done through the module info's `get_input()` method. In the case of a Scipy matrix, here, it just provides us with the matrix as an attribute.

Then we do whatever our module is designed to do. At the end, we write the output data to the appropriate output directory. This should always be obtained using the `get_absolute_output_dir()` method of the module info, since Pimlico takes care of the exact location for you.

Most Pimlico datatypes provide a corresponding **writer**, ensuring that the output is written in the correct format for it to be read by the datatype's reader. When we leave the `with` block, in which we give the writer the data it needs, this output is written to disk.

Pipeline config

Pipeline config files are fully documented in *Pipeline config*. Refer to that for all the details of how modules can be used in pipelines.

Todo: This section is copied from *Pimlico module structure*. It needs to be re-written to provide more technical and comprehensive documentation of pipeline config. NB: config files are fully documented in *Pipeline config*, so this just covers how ModuleInfo relates to the config.

Our module is now ready to use and we can refer to it in a pipeline config file. We'll assume we've prepared a suitable Scipy sparse matrix earlier in the pipeline, available as the default output of a module called `matrix`. Then we can add section like this to use our new module:

```
[matrix]
...(Produces sparse matrix output)...

[factorize]
type=myproject.modules.nmf
components=300
input=matrix
```

Note that, since there's only one input, we don't need to give its name. If we had defined multiple inputs, we'd need to specify this one as `input_matrix=matrix`.

You can now run the module as part of your pipeline in the usual ways.

1.2.5 Datatypes

A core concept in Pimlico is the *datatype*. All inputs and outputs to modules are associated with a datatype and typechecking ensures that outputs from one module are correctly matched up with inputs to the next.

Datatypes also provide interfaces for reading and writing datasets. They provide different ways of reading in or iterating over datasets and different ways to write out datasets, as appropriate to the datatype. They are used by Pimlico to typecheck connections between modules to make sure that the output from one module provides a suitable type of data for the input to another. They are then also used by the modules to read in their input data coming from earlier in a pipeline and to write out their output data, to be passed to later modules.

As much as possible, Pimlico pipelines should use *`standard datatypes`* to connect up the output of modules with the input of others. Most datatypes have a lot in common, which should be reflected in their sharing common base classes. **Input modules** take care of reading in data from external sources and they provide access to that data in a way that is identified by a Pimlico datatype.

Class structure

Instances of subclasses of *`PimlicoDatatype`* represent the type of datasets and are used for typechecking in a pipeline. Each datatype has an associated `Reader` class, accessed by `datatype_cls.Reader`. These are created automatically and can be instantiated via the datatype instance (by calling it). They are all subclasses of *`PimlicoDatatype`* `.Reader`.

It is these readers that are used within a pipeline to read a dataset output by an earlier module. In some cases, other readers may be used: for example, input modules provide standard datatypes at their outputs, but use special readers to provide access to the external data via the same interface as if the data had been stored within the pipeline.

A similar reflection of the datatype hierarchy is used for **dataset writers**, which are used to write the outputs from modules, to be passed to subsequent modules. These are created automatically, just like readers, and are all subclasses

of *PimlicoDatatype* .Writer. You can get a datatype's standard writer class via datatype_cls.Writer. Some datatypes might not provide a writer, but most do.

Note that you do not need to subclass or instantiate Reader, Writer or Setup classes yourself: subclasses are created automatically to correspond to each reader type. You can, however, add functionality to any of them by defining a nested class of the same name. It will automatically inherit from the parent datatype's corresponding class.

Readers

Most of the time, you don't need to worry about the process of getting hold of a reader, as it is done for you by the module. From within a module executor, you will usually do this:

```
reader = self.info.get_input("input_name")
```

reader is an instance of the datatype's Reader class, or some other reader class providing the same interface. You can use it to access the data, for example, iterating over a corpus, or reading in a file, depending on the datatype.

The follow guides describe the process that goes on internally in more detail.

Reader creation

The process of instantiating a reader for a given datatype is as follows:

1. Instantiate the datatype. A datatype instance is always associated with a module input (or output), so you rarely need to do this explicitly.
2. Use the datatype to instantiate a **reader setup**, by calling it.
3. Use the reader setup to check that the data is ready to reader by calling ready_to_read().
4. If the data is ready, use the reader setup to instantiate a reader, by calling it.

Reader setup

Reader setup classes provide any functionality relating to a reader needed before it is ready to read and instantiated. Like readers and writers, they are created automatically, so every Reader class has a Setup nested class.

Most importantly, the setup instance provides the ready_to_read() method, which indicates whether the reader is ready to be instantiated.

The standard implementation, which can be used in almost all cases, takes a list of possible paths to the dataset at initialization and checks whether the dataset is ready to be read from any of them. You generally don't need to override ready_to_read() with this, but just data_ready(path), which checks whether the data is ready to be read in a specific location. You can call the parent class' data-ready checks using super: super(MyDatatype.Reader.Setup, self).data_ready(path).

The whole Setup object will be passed to the corresponding Reader's init, so that it has access to data locations, etc. It can then be accessed as reader.setup.

Subclasses may take different init args/kwargs and store whatever attributes are relevant for preparing their corresponding Reader. In such cases, you will usually override a ModuleInfo's get_output_reader_setup() method for a specific output's reader preparation, to provide it with the appropriate arguments. Do this by calling the Reader class' get_setup(*args, **kwargs) class method, which passes args and kwargs through to the Setup's init.

You can add functionality to a reader's setup by creating a nested `Setup` class. This will inherit from the parent reader's setup. This happens automatically – you don't need to do it yourself and shouldn't inherit from anything. For example:

```
class MyDatatype(PimlicoDatatype):
    class Reader:
        # Overide reader things here

        class Setup:
            # Override setup things here
            # E.g.:
            def data_ready(path):
                # Parent checks: usually you want to do this
                if not super(MyDatatype.Reader.Setup, self).data_ready(path):
                    return False
                # Check whether the data's ready according to our own criteria
                # ...
                return True
```

Instantiate a reader setup of the relevant type by calling the datatype. Args and kwargs will be passed through to the `Setup` class' init. They may depend on the particular setup class, but typically one arg is required, which is a list of paths where the data may be found.

Reader from setup

You can use the reader setup to get a reader, once the data is ready to read.

This is done by simply calling the setup, with the pipeline instance as the first argument and, optionally, the name of the module that's currently being run. (If given, this will be used in error output, debugging, etc.)

The procedure then looks something like this:

```
datatype = ThisDatatype(options...)
# Most of the time, you will pass in a list of possible paths to the data
setup = datatype(possible_paths_list)
# Now check whether the data is ready to read
if setup.ready_to_read():
    reader = setup(pipeline, module="pipeline_module")
```

Creating a new datatype

This is the typical process for creating a new datatype. Of course, some datatypes do more, and some of the following is not always necessary, but it's a good guide for reference.

1. Create the datatype class, which may subclass *`PimlicoDatatype`* or some other existing datatype.
2. Specify a `datatype_name` as a class attribute.
3. Specify software dependencies for reading the data, if any, by overriding `get_software_dependencies()` (calling the super method as well).
4. Specify software dependencies for writing the data, if any that are not among the reading dependencies, by overriding `get_writer_software_dependencies()`.
5. Define a nested `Reader` class to add any methods to the reader for this datatype. The data should be read from the directory given by its `data_dir`. It should provide methods for getting different bits of the data, iterating over it, or whatever is appropriate.

6. Define a nested `Setup` class within the reader with a `data_ready(base_dir)` method to check whether the data in `base_dir` is ready to be read using the reader. If all that this does is check the existence of particular filenames or paths within the data dir, you can instead implement the `Setup` class' `get_required_paths()` method to return the paths relative to the data dir.
7. Define a nested `Writer` class in the datatype to add any methods to the writer for this datatype. The data should be written to the path given by its `data_dir`. Provide methods that the user can call to write things to the dataset. Required elements of the dataset should be specified as a list of strings as the `required_tasks` attribute and ticked off as written using `task_complete()`
8. You may want to specify:
   - `datatype_options`: an OrderedDict of option definitions
   - `shell_commands`: a list of shell commands associated with the datatype

Defining reader functionality

Naturally, different datatypes provide different ways to access their data. You do this by (implicitly) overriding the datatype's `Reader` class and adding methods to it.

As with `Setup` and `Writer` classes, you do not need to subclass the `Reader` explicitly yourself: subclasses are created automatically to correspond to each datatype. You add functionality to a datatype's reader by creating a nested `Reader` class, which inherits from the parent datatype's reader. This happens automatically – your nested class shouldn't inherit from anything:

```
class MyDatatype(PimlicoDatatype):
    class Reader:
        # Override reader things here
        def get_some_data(self):
            # Do whatever you need to do to provide access to the dataset
            # You probably want to use the attribute 'data_dir' to retrieve files
            # For example:
            with open(os.path.join(self.data_dir, "my_file.txt")) as f:
                some_data = f.read()
            return some_data
```

1.2.6 Module dependencies

In a Pimlico pipeline, you typically use lots of different external software packages. Some are Python packages, others system tools, Java libraries, whatever. Even the core modules that core with Pimlico between them depend on a huge amount of software.

Naturally, we don't want to have to install *all* of this software before you can run even a simple Pimlico pipeline that doesn't use all (or any) of it. So, we keep the core dependencies of Pimlico to an absolute minimum, and then check whether the necessary software dependencies are installed each time a pipeline module is going to be run.

Core dependencies

Certain dependencies are required for Pimlico to run at all, or needed so often that you wouldn't get far without installing them. These are defined in *pimlico.core.dependencies.core*, and when you run the Pimlico command-line interface, it checks they're available and tries to install them if they're not.

Module dependencies

Each module type defines its own set of software dependencies, if it has any. When you try to run the module, Pimlico runs some checks to try to make sure that all of these are available.

If some of them are not, it may be possible to install them automatically, straight from Pimlico. In particular, many Python packages can be very easily installed using Pip. If this is the case for one of the missing dependencies, Pimlico will tell you in the error output, and you can install them using the `install` command (with the module name/number as an argument).

Virtualenv

In order to simplify automatic installation, Pimlico is always run within a virtual environment, using Virtualenv. This means that any Python packages installed by Pip will live in a local directory within the Pimlico codebase that you're running and won't interfere with anything else on your system.

When you run Pimlico for the first time, it will create a new virtualenv for this purpose. Every time you run it after that, it will use this same environment, so anything you install will continue to be available.

Custom virtualenv

Most of the time, you don't even need to be aware of the virtualenv that Python's running in[1]. Under certain circumstances, you might need to use a custom virtualenv.

For example, say you're running your pipeline over different servers, but have the pipeline and Pimlico codebase on a shared network drive. Then you can find that the software installed in the virtualenv on one machine is incompatible with the system-wide software on the other.

You can specify a name for a custom virtualenv using the environment variable `PIMENV`. The first time you run Pimlico with this set, it will automatically create the new virtualenv.

```
$ PIMENV=myenv ./pimlico.sh mypipeline.conf status
```

Replace `myenv` with a name that better reflects its use (e.g. name of the server).

Every time you run Pimlico on that server, set the `PIMENV` environment variable in the same way.

In case you want to get to the virtualenv itself, you can find it in `pimlico/lib/virtualenv/myenv`.

Note: Pimlico previously used another environment variable `VIRTUALENV`, which gave a path to the virtualenv. You can still use this, but, unless you have a good reason to, it's easier to use `PIMENV`.

Defining module dependencies

Todo: Describe how module dependencies are defined for different types of deps

---

[1] If you're interested, it lives in `pimlico/lib/virtualenv/default`

Some examples

Todo: Include some examples from the core modules of how deps are defined and some special cases of software fetching

1.2.7 Local configuration

As well as knowing about the pipeline you're running, Pimlico also needs to know some things about the setup of the system on which you're running it. This is completely independent of the pipeline config: the same pipeline can be run on different systems with different local setups.

A couple of settings must always be provided for Pimlico: the **long-term** and **short-term stores** (see *Data stores* below). Other system settings may be specified as necessary. (At the time of writing, there aren't any, but they will be documented here as they arise.) See *Other Pimlico settings* below.

Specific modules may also have system-level settings. For example, a module that calls an external tool may need to know the location of that tool, or how much memory it can use on this system. Any that apply to the built-in Pimlico modules are listed below in *Settings for built-in modules*.

Local config file location

Pimlico looks in various places to find the local config settings. Settings are loaded in a particular order, overriding earlier versions of the same setting as we go (see *pimlico.core.config.PipelineConfig.load_local_config()*).

Settings are specified with the following order of precedence (those later override the earlier):

```
local config file < host-specific config file < cmd-line overrides
```

Most often, you'll just specify all settings in the main local config file. This is a file in your home directory named `.pimlico`. This must exist for Pimlico to be able to run at all.

Host-specific config

If you share your home directory between different computers (e.g. a networked filesystem), the above setup could cause a problem, as you may need a different local config on the different computers. Pimlico allows you to have special config files that only get read on machines will a particular hostname.

For example, say I have two computers, `localbox` and `remotebox`, which share a home directory. I've created my `.pimlico` local config file on `localbox`, but need to specify a different storage location on `remotebox`. I simply create another config file called `.pimlico_remotebox``[#hostname]_. Pimlico will load first the basic local config in ``.pimlico` and then override those settings with what it reads from the host-specific config file.

You can also specify a hostname prefix to match. Say I've got a whole load of computers I want to be able to run on, with hostnames `remotebox1`, `remotebox2`, etc. If I create a config file called `.pimlico_remotebox-`, it will be used on all of these hosts.

Command-line overrides

Occasionally, you might want to specify a local config setting just for one run of Pimlico. Use the `--override-local-config` (or `-l`) to specify a value for an individual setting in the form `setting=value`. For example:

```
./pimlico.sh mypipeline.conf -l somesetting=5 run mymodule
```

If you want to override multiple settings, simply use the option multiple times.

Custom location

If the above solutions don't work for you, you can also explicitly specify on the command line an alternative location from which to load the local config file that Pimlico typically expects to find in `~/.pimlico`.

Use the `--local-config` parameter to give a filename to use instead of the `~/.pimlico`.

For example, if your home directory is shared across servers and the above hostname-specific config solution doesn't work in your case, you can fall back to pointing Pimlico at your own host-specific config file.

Data stores

Pimlico needs to know where to put and find output files as it executes. Settings are given in the local config, since they apply to all Pimlico pipelines you run and may vary from system to system. Note that Pimlico will make sure that different pipelines don't interfere with each other's output (provided you give them different names): all pipelines store their output and look for their input within these same base locations.

See *Data storage* for an explanation of Pimlico's data store system.

At least one store must be given in the local config:

```
store=/path/to/storage/root
```

You may specify as many storage locations as you like, giving each a name:

```
store_fast=/path/to/fast/store
store_big=/path/to/big/store
```

If you specify named stores *and* an unnamed one, the unnamed one will be used as the default output store. Otherwise, the first in the file will be the default.

```
store=/path/to/a/store          # This will be the default output store
store_fast=/path/to/fast/store  # These will be additional, named stores
store_big=/path/to/big/store
```

Other Pimlico settings

In future, there will no doubt be more settings that you can specify at the system level for Pimlico. These will be documented here as they arise.

Settings for built-in modules

Specific modules may consult the local config to allow you to specify settings for them. We cannot document them here for all modules, as we don't know what modules are being developed outside the core codebase. However, we can provide a list here of the settings consulted by built-in Pimlico modules.

There aren't any yet, but they will be listed here as they arise.

Footnotes:

1.2.8 Data storage

Pimlico needs to know where to put and find output files as it executes, in order to store data and pass it between modules. On any particular system running Pimlico, multiple locations (**stores**) may be used as storage and Pimlico will check all of them when it's looking for a module's data.

Single store

Let's start with a simple setup with just one store. A setting `store` in the local config (see *Local configuration*) specifies the root directory of this store. This applies to all Pimlico pipelines you run on this system and Pimlico will make sure that different pipelines don't interfere with each other's output (provided you give them different names).

When you run a pipeline module, its output will be stored in a subdirectory specific to that pipeline and that module with the store's root directory. When Pimlico needs to use that data as input to another module, it will look in the appropriate directory within the store.

Multiple stores

For various reasons, you may wish to store Pimlico data in multiple locations.

For example, one common scenario is that you have access to a disk that is fast to write to (call it *fast-disk*), but not very big, and another disk (e.g. over a network filesystem) that has lots of space, but is slower (call it *big-disk*). You therefore want Pimlico to output its data, much of which might only be used fleetingly and then no longer needed, to *fast-disk*, so the processing runs quickly. Then, you want to move the output from certain modules over to *big-disk*, to make space on *fast-disk*.

We can define two stores for Pimlico to use and give them names. The first ("fast") will be used to output data to (just like the sole store in the previous section). The second ("big"), however, will also be checked for module data, meaning that we can move data from "fast" to "big" whenever we like.

Instead of using the `store` parameter in the local config, we use multiple `store_<name>` parameters. One of them (the first one, or the one given by `store` with no name, if you include that) we be treated as the default output store.

Specific the locations in the local config like this:

```
store_fast=/path/to/fast/store
store_big=/path/to/big/store
```

Remember, these paths are not specific to a pipeline: all pipelines will use different subdirectories of these ones.

To check what stores you've got in your current configuration, use the *stores* command.

Moving data between stores

Say you've got a two-store setup like in the previous example. You've now run a module that produces a lot of output and want to move it to your big disk and have Pimlico read it from there.

You don't need to replicate the directory structure yourself and move module output between stores. Pimlico has a command *movestores* to do this for you. Specify the name of the store you want to move data to (`big` in this case) and the names or numbers of the modules whose data you want to move.

Once you've done that, Pimlico should continue to behave as it did before, just as if the data was still in its original location.

Updating from the old storage system

Prior to v0.8, Pimlico used a different system of storage locations. If you have a local config file (`~/.pimlico`) from an earlier version you will see deprecation warnings.

Change something like this:

```
long_term_store=/path/to/long/store
short_term_store=/path/to/short/store
```

to something like this:

```
store_long=/path/to/long/store
store_short=/path/to/short/store
```

Or, if you only ever needed one storage location, simply this:

```
store=/path/to/store
```

1.2.9 Python scripts

All the heavy work of your data-processing is implemented in Pimlico modules, either by loading core Pimlico modules from your pipeline config file or by writing your own modules. Sometimes, however, it can be handy to write a quick Python script to get hold of the output of one of your pipeline's modules and inspect it or do something with it.

This can be easily done writing a Python script and using the *python* shell command to run it. This command loads your pipeline config (just like all others) and then either runs a script you've specified on the command line, or enters an interactive Python shell. The advantages of this over just running the normal `python` command on the command line are that the script is run in the same execution context used for your pipeline (e.g. using the Pimlico instance's virtualenv) and that the loaded pipeline is available to you, so you can easily can hold of its data locations, datatypes, etc.

Accessing the pipeline

At the top of your Python script, you can get hold of the loaded pipeline config instance like this:

```
from pimlico.cli.pyshell import get_pipeline

pipeline = get_pipeline()
```

Now you can use this to get to, among other things, the pipeline's modules and their input and output datasets. A module called `module1` can be accessed by treating the pipeline like a dict:

```
module = pipeline["module1"]
```

This gives you the `ModuleInfo` instance for that module, giving access to its inputs, outputs, options, etc:

```
data = module.get_output("output_name")
```

Writing and running scripts

All of the above code to access a pipeline can be put in a Python script somewhere in your codebase and run from the command line. Let's say I create a script `src/python/scripts/myscript.py` containing:

```
from pimlico.cli.pyshell import get_pipeline

pipeline = get_pipeline()
module = pipeline["module1"]
data = module.get_output("output_name")
# Here we can start probing the data using whatever interface the datatype provides
print data
```

Now I can run this from the root directory of my project as follows:

```
./pimlico.sh mypipeline.conf python src/python/scripts/myscript.py
```

1.3 Core Pimlico modules

Pimlico comes with a substantial collection of module types that provide wrappers around existing NLP and machine learning tools, as well as a number of general tools for processing datasets that are useful for many applications.

Some modules that used to be among the core modules have not yet been updated since a big change in the datatypes system. They can be found in `pimlico.old_datatypes.modules`, but are not currently functional, until I get round to updating them.

1.3.1 Corpus manipulation

Core modules for generic manipulation of mainly iterable corpora.

Corpus concatenation

Path	pimlico.modules.corpora.concat
Executable	no

Concatenate two (or more) corpora to produce a bigger corpus.

They must have the same data point type, or one must be a subtype of the other.

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
corpora	*list* of *iterable_corpus*

Outputs

Name	Type(s)
corpus	*corpus with data-point from input*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_concat_module]
type=pimlico.modules.corpora.concat
input_corpora=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *concat*

Corpus statistics

Path	pimlico.modules.corpora.corpus_stats
Executable	yes

Some basic statistics about tokenized corpora

Counts the number of tokens, sentences and distinct tokens in a corpus.

Inputs

Name	Type(s)
corpus	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
stats	*named_file*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_corpus_stats_module]
type=pimlico.modules.corpora.corpus_stats
input_corpus=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *stats*

Human-readable formatting

Path	pimlico.modules.corpora.format
Executable	yes

Corpus formatter

Pimlico provides a data browser to make it easy to view documents in a tarred document corpus. Some datatypes provide a way to format the data for display in the browser, whilst others provide multiple formatters that display the data in different ways.

This module allows you to use this formatting functionality to output the formatted data as a corpus. Since the formatting operations are designed for display, this is generally only useful to output the data for human consumption.

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
formatted	*grouped_corpus* <*RawTextDocumentType*>

Options

Name	Description	Type
formatter	Fully qualified class name of a formatter to use to format the data. If not specified, the default formatter is used, which uses the datatype's browser_display attribute if available, or falls back to just converting documents to unicode	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_format_module]
type=pimlico.modules.corpora.format
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_format_module]
type=pimlico.modules.corpora.format
input_corpus=module_a.some_output
formatter=path.to.formatter.FormatterClass
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *interleave*
- *subset*
- *concat*
- *group*
- *tokenized_formatter*

Archive grouper (filter)

Path	pimlico.modules.corpora.group
Executable	no

Group the data points (documents) of an iterable corpus into fixed-size archives. This is a standard thing to do at the start of the pipeline, since it's a handy way to store many (potentially small) files without running into filesystem problems.

The documents are simply grouped linearly into a series of groups (archives) such that each (apart from the last) contains the given number of documents.

After grouping documents in this way, document map modules can be called on the corpus and the grouping will be preserved as the corpus passes through the pipeline.

Note: This module used to be called `tar_filter`, but has been renamed in keeping with other changes in the new datatype system.

There also used to be a `tar` module that wrote the grouped corpus to disk. This has now been removed, since most of the time it's fine to use this filter module instead. If you really want to store the grouped corpus, you can use the `store` module.

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
documents	*iterable_corpus*

Outputs

Name	Type(s)
documents	*grouped corpus with input doc type*

Options

Name	Description	Type
archive_basename	Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Number of documents to include in each archive (default: 1k)	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_group_module]
type=pimlico.modules.corpora.group
input_documents=module_a.some_output
```

This example usage includes more options.

```
[my_group_module]
type=pimlico.modules.corpora.group
input_documents=module_a.some_output
archive_basename=archive
archive_size=1000
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *group*
- *store*

Interleaved corpora

Path	pimlico.modules.corpora.interleave
Executable	no

Interleave data points from two (or more) corpora to produce a bigger corpus.

Similar to *concat*, but interleaves the documents when iterating. Preserves the order of documents within corpora and takes documents two each corpus in inverse proportion to its length, i.e. spreads out a smaller corpus so we don't finish iterating over it earlier than the longer one.

They must have the same data point type, or one must be a subtype of the other.

In theory, we could find the most specific common ancestor and use that as the output type, but this is not currently implemented and may not be worth the trouble. Perhaps we will add this in future.

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
corpora	*list* of *grouped_corpus*

Outputs

Name	Type(s)
corpus	*grouped corpus with input doc type*

Options

Name	Description	Type
archive_basename	Documents are regrouped into new archives. Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Documents are regrouped into new archives. Number of documents to include in each archive (default: 1k)	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_interleave_module]
type=pimlico.modules.corpora.interleave
input_corpora=module_a.some_output
```

This example usage includes more options.

```
[my_interleave_module]
type=pimlico.modules.corpora.interleave
input_corpora=module_a.some_output
archive_basename=archive
archive_size=1000
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *interleave*

Corpus document list filter

Path	pimlico.modules.corpora.list_filter
Executable	yes

Similar to `split`, but instead of taking a random split of the dataset, splits it according to a given list of documents, putting those in the list in one set and the rest in another.

Inputs

Name	Type(s)
corpus	*grouped_corpus*
list	*string_list*

Outputs

Name	Type(s)
set1	*grouped corpus with input doc type*
set2	*grouped corpus with input doc type*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_list_filter_module]
type=pimlico.modules.corpora.list_filter
input_corpus=module_a.some_output
input_list=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *list_filter*

Random shuffle

Path	pimlico.modules.corpora.shuffle
Executable	yes

Randomly shuffles all the documents in a grouped corpus, outputting them to a new set of archives with the same sizes as the input archives.

This was difficult to do this efficiently for a large corpus using the old tar storage format. There therefore used to be a strategy implemented here where the input documents were read in linear order and placed into a temporary set of small archives ("bins") and these were concatenated into the larger archives, shuffling the documents in memory in each during the process.

It is no longer necessary to do this, since the standard pipeline-internal storage format permits efficient random access. However, it may sometimes be necessary to use the linear-reading strategy: for example, if the input comes from a filter module, its documents cannot be randomly accessed.

Todo: Currently, this accepts any GroupedCorpus as input, but checks at runtime that the input is stored used the pipeline-internal format. It would be much better if this check could be enforced at the level of datatypes, so that the input datatype requirement explicitly rules out grouped corpora coming from input readers, filters or other dynamic sources.

Since this requires some tricky changes to the datatype system, I'm not implementing it now, but it should be done in future.

It will be implemented as part of the replacement of `GroupedCorpus` by `StoredIterableCorpus`: **'https://github.com/markgw/pimlico/issues/24'_**

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
corpus	*grouped corpus with input doc type*

Options

Name	Description	Type
archive_basename	Basename to use for archives in the output corpus. Default: 'archive'	string
seed	Seed for the random number generator. The RNG is always seeded, for reproducibility. Default: 999	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_shuffle_module]
type=pimlico.modules.corpora.shuffle
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_shuffle_module]
type=pimlico.modules.corpora.shuffle
input_corpus=module_a.some_output
archive_basename=archive
seed=999
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *shuffle*

Random shuffle (linear)

Path	pimlico.modules.corpora.shuffle_linear
Executable	yes

Randomly shuffles all the documents in a grouped corpus, outputting them to a new set of archives with the same sizes as the input archives.

It is difficult to do this efficiently for a large corpus when we cannot randomly access the input documents. Under the old, now deprecated, tar-based storage format, random access was costly. If a corpus is produced on the fly, e.g. from a filter or input reader, random access is impossible.

We use a strategy where the input documents are read in linear order and placed into a temporary set of small archives ("bins"). Then these are concatenated into the larger archives, shuffling the documents in memory in each during the process.

The expected average size of the temporary bins can be set using the `bin_size` parameter. Alternatively, the exact total number of bins to use can be set using the `num_bins` parameter.

It may be necessary to lower the bin size if, for example, your individual documents are very large files. You might also find the process is noticeably faster with a higher bin size if your files are small.

See also:

Module type *`pimlico.modules.corpora.shuffle`* If the input corpus is not dynamically produced and is therefore randomly accessible, it is more efficient to use the `shuffle` module type.

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
corpus	*grouped corpus with input doc type*

Options

Name	Description	Type
archive_basename	Basename to use for archives in the output corpus. Default: 'archive'	string
bin_size	Target expected size of temporary bins into which documents are shuffled. The actual size may vary, but they will on average have this size. Default: 100	int
keep_archive_names	By default, it is assumed that all doc names are unique to the whole corpus, so the same doc names are used once the documents are put into their new archives. If doc names are only unique within the input archives, use this and the input archive names will be included in the output document names. Default: False	bool
num_bins	Directly set the number of temporary bins to put document into. If set, bin_size is ignored	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_shuffle_module]
type=pimlico.modules.corpora.shuffle_linear
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_shuffle_module]
type=pimlico.modules.corpora.shuffle_linear
input_corpus=module_a.some_output
archive_basename=archive
bin_size=100
keep_archive_names=F
num_bins=0
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *shuffle*

Corpus split

Path	pimlico.modules.corpora.split
Executable	yes

Split a tarred corpus into two subsets. Useful for dividing a dataset into training and test subsets. The output datasets have the same type as the input. The documents to put in each set are selected randomly. Running the module multiple times will give different splits.

Note that you can use this multiple times successively to split more than two ways. For example, say you wanted a training set with 80% of your data, a dev set with 10% and a test set with 10%, split it first into training and non-training 80-20, then split the non-training 50-50 into dev and test.

The module also outputs a list of the document names that were included in the first set. Optionally, it outputs the same thing for the second input too. Note that you might prefer to only store this list for the smaller set: e.g. in a

training-test split, store only the test document list, as the training list will be much larger. In such a case, just put the smaller set first and don't request the optional output *doc_list2*.

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
set1	*grouped corpus with input doc type*
set2	*grouped corpus with input doc type*
doc_list1	*string_list*

Optional

Name	Type(s)
doc_list2	*string_list*

Output groups

The module defines some named output groups, which can be used to refer to collections of outputs at once, as multiple inputs to another module or alternative inputs.

Group name	Outputs
corpora	set1, set2

Options

Name	Description	Type
set1_size	Proportion of the corpus to put in the first set, float between 0.0 and 1.0. If an integer >1 is given, this is treated as the absolute number of documents to put in the first set, rather than a proportion. Default: 0.2 (20%)	float

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_split_module]
type=pimlico.modules.corpora.split
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_split_module]
type=pimlico.modules.corpora.split
input_corpus=module_a.some_output
set1_size=0.20
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *split*

Store a corpus

Path	pimlico.modules.corpora.store
Executable	yes

Store a corpus

Take documents from a corpus and write them to disk using the standard writer for the corpus' data point type. This is useful where documents are produced on the fly, for example from some filter module or from an input reader, but where it is desirable to store the produced corpus for further use, rather than always running the filters/readers each time the corpus' documents are needed.

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
corpus	*grouped corpus with input doc type*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_store_module]
type=pimlico.modules.corpora.store
input_corpus=module_a.some_output
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *store*
- *filter_tokenize*
- *europarl*
- *raw_text_files_test*
- *filter_map*

Random subsample

Path	pimlico.modules.corpora.subsample
Executable	yes

Randomly subsample documents of a corpus at a given rate to create a smaller corpus.

Inputs

Name	Type(s)
corpus	*grouped_corpus*

Outputs

Name	Type(s)
corpus	*corpus with data-point from input*

Options

Name	Description	Type
p	(required) Probability of including any given document. The resulting corpus will be roughly this proportion of the size of the input. Alternatively, you can specify an integer, which will be interpreted as the target size of the output. A p value will be calculated based on the size of the input corpus	float
seed	Random seed. We always set a random seed before starting to ensure some level of reproducability	int
skip_invalid	Skip over any invalid documents so that the output subset contains just valid document and no invalid ones. By default, invalid documents are passed through	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_subsample_module]
type=pimlico.modules.corpora.subsample
input_corpus=module_a.some_output
p=0.1
```

This example usage includes more options.

```
[my_subsample_module]
type=pimlico.modules.corpora.subsample
input_corpus=module_a.some_output
p=0.1
seed=1234
skip_invalid=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *subsample*

Corpus subset

Path	pimlico.modules.corpora.subset
Executable	no

Simple filter to truncate a dataset after a given number of documents, potentially offsetting by a number of documents. Mainly useful for creating small subsets of a corpus for testing a pipeline before running on the full corpus.

Can be run on an iterable corpus or a tarred corpus. If the input is a tarred corpus, the filter will emulate a tarred corpus with the appropriate datatype, passing through the archive names from the input.

When a number of valid documents is required (calculating corpus length when skipping invalid docs), if one is stored in the metadata as `valid_documents`, that count is used instead of iterating over the data to count them up.

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
corpus	*iterable_corpus*

Outputs

Name	Type(s)
corpus	*corpus with data-point from input*

Options

Name	Description	Type
offset	Number of documents to skip at the beginning of the corpus (default: 0, start at beginning)	int
size	(required) Number of documents to include	int
skip_invalid	Skip over any invalid documents so that the output subset contains the chosen number of (valid) documents (or as many as possible) and no invalid ones. By default, invalid documents are passed through and counted towards the subset size	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_subset_module]
type=pimlico.modules.corpora.subset
input_corpus=module_a.some_output
size=100
```

This example usage includes more options.

```
[my_subset_module]
type=pimlico.modules.corpora.subset
input_corpus=module_a.some_output
offset=0
size=100
skip_invalid=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *subset*

Corpus vocab builder

Path	pimlico.modules.corpora.vocab_builder
Executable	yes

Builds a dictionary (or vocabulary) for a tokenized corpus. This is a data structure that assigns an integer ID to every distinct word seen in the corpus, optionally applying thresholds so that some words are left out.

Similar to `pimlico.modules.features.vocab_builder`, which builds two vocabs, one for terms and one for features.

May specify a list of stopwords, which will be ignored, even if they're found in the corpus. The filter to remove frequent words (controlled by *max_prop*) will potentially add further stopwords, so the resulting list is output as *stopwords*.

Inputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
vocab	*dictionary*
stopwords	*string_list*

Options

Name	Description	Type
include	Ensure that certain words are always included in the vocabulary, even if they don't make it past the various filters, or are never seen in the corpus. Give as a comma-separated list	comma-separated list of strings
limit	Limit vocab size to this number of most common entries (after other filters)	int
max_prop	Include terms that occur in max this proportion of documents	float
oov	Represent OOVs using the given string in the vocabulary. Used to represent chars that will be out of vocabulary after applying threshold/limit filters. Included in the vocabulary even if the count is 0	string
prune_at	Prune the dictionary if it reaches this size. Setting a lower value avoids getting stuck with too big a dictionary to be able to prune and slowing things down, but means that the final pruning will less accurately reflect the true corpus stats. Should be considerably higher than limit (if used). Set to 0 to disable. Default: 2M (Gensim's default)	int
threshold	Minimum number of occurrences required of a term to be included	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_vocab_builder_module]
type=pimlico.modules.corpora.vocab_builder
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_vocab_builder_module]
type=pimlico.modules.corpora.vocab_builder
input_text=module_a.some_output
include=word1,word2,...
limit=10k
oov=text
prune_at=2000000
threshold=100
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*
- *custom_module_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *vocab_builder*

Token frequency counter

Path	pimlico.modules.corpora.vocab_counter
Executable	yes

Count the frequency of each token of a vocabulary in a given corpus (most often the corpus on which the vocabulary was built).

Note that this distribution is not otherwise available along with the vocabulary. It stores the document frequency counts - how many documents each token appears in - which may sometimes be a close enough approximation to the actual frequencies. But, for example, when working with character-level tokens, this estimate will be very poor.

The output will be a 1D array whose size is the length of the vocabulary, or the length plus one, if `oov_excluded=T` (used if the corpus has been mapped so that OOVs are represented by the ID `vocab_size+1`, instead of having a special token).

Inputs

Name	Type(s)
corpus	*grouped_corpus*<*IntegerListsDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
distribution	*numpy_array*

Options

Name	Description	Type
oov_excluded	Indicates that the corpus has been mapped so that OOVs are represented by the ID vocab_size+1, instead of having a special token in the vocab	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_vocab_counter_module]
type=pimlico.modules.corpora.vocab_counter
input_corpus=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_vocab_counter_module]
type=pimlico.modules.corpora.vocab_counter
input_corpus=module_a.some_output
input_vocab=module_a.some_output
oov_excluded=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *vocab_counter*

Tokenized corpus to ID mapper

Path	pimlico.modules.corpora.vocab_mapper
Executable	yes

Maps all the words in a tokenized textual corpus to integer IDs, storing just lists of integers in the output.

This is typically done before doing things like training models on textual corpora. It ensures that a consistent mapping from words to IDs is used throughout the pipeline. The training modules use this pre-mapped form of input, instead of performing the mapping as they read the data, because it is much more efficient if the corpus needs to be iterated over many times, as is typical in model training.

First use the *vocab_builder* module to construct the word-ID mapping and filter the vocabulary as you wish, then use this module to apply the mapping to the corpus.

Inputs

Name	Type(s)
text	*grouped_corpus* <*TokenizedDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
ids	*grouped_corpus* <*IntegerListsDocumentType*>

Options

Name	Description	Type
oov	If given, special token to map all OOV tokens to. Otherwise, use vocab_size+1 as index. Special value 'skip' simply skips over OOV tokens	string
row_length_bytes	The length of each row is stored, by default, using a 2-byte value. If your dataset contains very long lines, you can increase this to allow larger row lengths to be stored	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_vocab_mapper_module]
type=pimlico.modules.corpora.vocab_mapper
input_text=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_vocab_mapper_module]
type=pimlico.modules.corpora.vocab_mapper
input_text=module_a.some_output
input_vocab=module_a.some_output
oov=value
row_length_bytes=2
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *vocab_mapper*
- *vocab_mapper*

ID to tokenized corpus mapper

Path	pimlico.modules.corpora.vocab_unmapper
Executable	yes

Maps all the IDs in an integer lists corpus to their corresponding words in a vocabulary, producing a tokenized textual corpus.

This is the inverse of `vocab_mapper`, which maps words to IDs. Typically, the resulting integer IDs are used for model training, but sometimes we need to map in the opposite direction.

Inputs

Name	Type(s)
ids	*grouped_corpus*<*IntegerListsDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
oov	If given, assume the vocab_size+1 was used to represent out-of-vocabulary words and map this index to the given token. Special value 'skip' simply skips over vocab_size+1 indices	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_vocab_unmapper_module]
type=pimlico.modules.corpora.vocab_unmapper
input_ids=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_vocab_unmapper_module]
type=pimlico.modules.corpora.vocab_unmapper
input_ids=module_a.some_output
input_vocab=module_a.some_output
oov=value
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *vocab_unmapper*

1.3.2 Embeddings

Modules for extracting features from which to learn word embeddings from corpora, and for training embeddings.

Some of these don't actually learn the embeddings, they just produce features which can then be fed into an embedding learning module, such as a form of matrix factorization. Note that you can train embeddings not only using the trainers here, but also using generic matrix manipulation techniques, for example the factorization methods provided by sklearn.

fastText embedding trainer

Path	pimlico.modules.embeddings.fasttext
Executable	yes

Train fastText embeddings on a tokenized corpus.

Uses the *fastText Python package <https://fasttext.cc/docs/en/python-module.html>*.

FastText embeddings store more than just a vector for each word, since they also have sub-word representations. We therefore store a standard embeddings output, with the word vectors in, and also a special fastText embeddings output.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
embeddings	*embeddings*
model	fasttext_embeddings

Options

Name	Description	Type
bucket	number of buckets. Default: 2,000,000	int
dim	size of word vectors. Default: 100	int
epoch	number of epochs. Default: 5	int
loss	loss function: ns, hs, softmax, ova. Default: ns	'ns', 'hs', 'softmax' or 'ova'
lr	learning rate. Default: 0.05	float
lr_update_rate	change the rate of updates for the learning rate. Default: 100	int
maxn	max length of char ngram. Default: 6	int
min_count	minimal number of word occurences. Default: 5	int
minn	min length of char ngram. Default: 3	int
model	unsupervised fasttext model: cbow, skipgram. Default: skipgram	'skipgram' or 'cbow'
neg	number of negatives sampled. Default: 5	int
t	sampling threshold. Default: 0.0001	float
verbose	verbose. Default: 2	int
word_ngrams	max length of word ngram. Default: 1	int
ws	size of the context window. Default: 5	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fasttext_module]
type=pimlico.modules.embeddings.fasttext
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_fasttext_module]
type=pimlico.modules.embeddings.fasttext
input_text=module_a.some_output
bucket=2000000
dim=100
epoch=5
loss=ns
lr=0.05
lr_update_rate=100
maxn=6
min_count=5
minn=3
model=skipgram
neg=5
t=0.00
verbose=2
word_ngrams=1
ws=5
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *fasttext_train*

GloVe embedding trainer

Path	pimlico.modules.embeddings.glove
Executable	yes

Train GloVe embeddings on a tokenized corpus.

Uses the *original GloVe code <https://github.com/stanfordnlp/GloVe>*, called in a subprocess.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
embeddings	*embeddings*
glove_output	named_file_collection

Options

Name	Description	Type
alpha	Parameter in exponent of weighting function; default 0.75	float
array_size	Limit to length <array_size> the buffer which stores chunks of data to shuffle before writing to disk. This value overrides that which is automatically produced by 'memory'	int
distance_weighting	If False, do not weight cooccurrence count by distance between words; if True (default), weight the cooccurrence count by inverse of distance between words	bool
eta	Initial learning rate; default 0.05	float
grad_clip	Gradient components clipping parameter. Values will be clipped to [-grad-clip, grad-clip] interval	float
iter	Number of training iterations; default 25	int
max_product	Limit the size of dense cooccurrence array by specifying the max product of the frequency counts of the two cooccurring words. This value overrides that which is automatically produced by 'memory'. Typically only needs adjustment for use with very large corpora.	int
max_vocab	Upper bound on vocabulary size, i.e. keep the <max_vocab> most frequent words. The minimum frequency words are randomly sampled so as to obtain an even distribution over the alphabet. Default: 0 (no limit)	int
memory	Soft limit for memory consumption, in GB – based on simple heuristic, so not extremely accurate; default 4.0	float
min_count	Lower limit such that words which occur fewer than <min_count> times are discarded. Default: 0	int
overflow_length	Limit to length the sparse overflow array, which buffers cooccurrence data that does not fit in the dense array, before writing to disk. This value overrides that which is automatically produced by 'memory'. Typically only needs adjustment for use with very large corpora.	int
seed	Random seed to use for shuffling. If not set, will be randomized using current time	int
symmetric	If False, only use left context; if True (default), use left and right	bool
threads	Number of threads during training; default 8	int
vector_size	Dimension of word vector representations (excluding bias term); default 50	int
window_size	Number of context words to the left (and to the right, if symmetric = 1); default 15	int
x_max	Parameter specifying cutoff in weighting function; default 100.0	float

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_glove_module]
type=pimlico.modules.embeddings.glove
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_glove_module]
type=pimlico.modules.embeddings.glove
input_text=module_a.some_output
alpha=0.75
array_size=0
distance_weighting=T
eta=0.05
grad_clip=0.1
iter=25
max_product=0
max_vocab=0
memory=4.00
min_count=0
overflow_length=0
seed=0
symmetric=T
threads=8
vector_size=50
window_size=15
x_max=100.00
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *glove_train*

Doc embedding mappers

Produce datatypes that can map tokens in documents to their embeddings.

fastText to doc-embedding mapper

Path	pimlico.modules.embeddings.mappers.fasttext
Executable	yes

Use trained fastText embeddings to map words to their embeddings, including OOVs, using sub-word information.

First train a fastText model using the fastText training module. Then use this module to produce a doc-embeddings mapper.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
embeddings	`fasttext_embeddings`

Outputs

Name	Type(s)
mapper	*fasttext_doc_embeddings_mapper*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fasttext_doc_mapper_module]
type=pimlico.modules.embeddings.mappers.fasttext
input_embeddings=module_a.some_output
```

Fixed embeddings to doc-embedding mapper

Path	pimlico.modules.embeddings.mappers.fixed
Executable	yes

Use trained fixed word embeddings to map words to their embeddings. Does nothing with OOVs, which we don't have any way to map.

First train or load embeddings using another module. Then use this module to produce a doc-embeddings mapper.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
embeddings	*embeddings*

Outputs

Name	Type(s)
mapper	*fixed_embeddings_doc_embeddings_mapper*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fixed_embeddings_doc_mapper_module]
type=pimlico.modules.embeddings.mappers.fixed
input_embeddings=module_a.some_output
```

Normalize embeddings

Path	pimlico.modules.embeddings.normalize
Executable	yes

Apply normalization to a set of word embeddings.

For now, only one type of normalization is provided: L2 normalization. Each vector is scaled so that its Euclidean magnitude is 1.

Other normalizations (like L1 or variance normalization) may be added in future.

Inputs

Name	Type(s)
embeddings	*embeddings*

Outputs

Name	Type(s)
embeddings	*embeddings*

Options

Name	Description	Type
l2_norm	Apply L2 normalization to scale each vector to unit length. Default: T	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_normalize_embeddings_module]
type=pimlico.modules.embeddings.normalize
input_embeddings=module_a.some_output
```

This example usage includes more options.

```
[my_normalize_embeddings_module]
type=pimlico.modules.embeddings.normalize
input_embeddings=module_a.some_output
l2_norm=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *embedding_norm*

Store embeddings (internal)

Path	pimlico.modules.embeddings.store_embeddings
Executable	yes

Simply stores embeddings in the Pimlico internal format.

This is not often needed, but can be useful if reading embeddings for an input reader that is slower than reading from the internal format. Then you can use this module to do the reading and store the result before passing it to other modules.

Inputs

Name	Type(s)
embeddings	*embeddings*

Outputs

Name	Type(s)
embeddings	*embeddings*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_store_embeddings_module]
type=pimlico.modules.embeddings.store_embeddings
input_embeddings=module_a.some_output
```

Store in TSV format

Path	pimlico.modules.embeddings.store_tsv
Executable	yes

Takes embeddings stored in the default format used within Pimlico pipelines (see *Embeddings*) and stores them as TSV files.

This is for using the vectors outside your pipeline, for example, for distributing them publicly or using as input to an external visualization tool. For passing embeddings between Pimlico modules, the internal *Embeddings* datatype should be used.

These are suitable as input to the Tensorflow Projector.

Inputs

Name	Type(s)
embeddings	*embeddings*

Outputs

Name	Type(s)
embeddings	*tsv_vec_files*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_store_tsv_module]
type=pimlico.modules.embeddings.store_tsv
input_embeddings=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *tsvvec_store*

Store in word2vec format

Path	pimlico.modules.embeddings.store_word2vec
Executable	yes

Takes embeddings stored in the default format used within Pimlico pipelines (see *Embeddings*) and stores them using the `word2vec` storage format.

This is for using the vectors outside your pipeline, for example, for distributing them publicly. For passing embeddings between Pimlico modules, the internal *Embeddings* datatype should be used.

The output contains a `bin` file, containing the vectors in the binary format, and a `vocab` file, containing the vocabulary and word counts.

Uses the Gensim implementation of the storage, so depends on Gensim.

Does not support Python 2, since we depend on Gensim.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
embeddings	*embeddings*

Outputs

Name	Type(s)
embeddings	*word2vec_files*

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_store_word2vec_module]
type=pimlico.modules.embeddings.store_word2vec
input_embeddings=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *word2vec_store*

Word2vec embedding trainer

Path	pimlico.modules.embeddings.word2vec
Executable	yes

Word2vec embedding learning algorithm, using Gensim's implementation.

Find out more about word2vec.

This module is simply a wrapper to call Gensim Python (+C)'s implementation of word2vec on a Pimlico corpus.

Does not support Python 2 since Gensim has dropped Python 2 support.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
model	*embeddings*

Options

Name	Description	Type
iters	number of iterations over the data to perform. Default: 5	int
min_count	word2vec's min_count option: prunes the dictionary of words that appear fewer than this number of times in the corpus. Default: 5	int
negative_samples	number of negative samples to include per positive. Default: 5	int
size	number of dimensions in learned vectors. Default: 200	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_word2vec_module]
type=pimlico.modules.embeddings.word2vec
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_word2vec_module]
type=pimlico.modules.embeddings.word2vec
input_text=module_a.some_output
iters=5
min_count=5
negative_samples=5
size=200
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *word2vec_train*

1.3.3 Gensim topic modelling

Modules providing access to topic model training and other routines from Gensim.

Topic model topic coherence

Path	pimlico.modules.gensim.coherence
Executable	yes

Compute topic coherence.

Takes input as a list of the top words for each topic. This can be produced from various types of topic model, so they can all be evaluated using this method.

Also requires a corpus from which to compute the PMI statistics. This should typically be a different corpus to that on which the model was trained.

For now, this just computes statistics and outputs them to a text file, and also outputs a single number representing the mean topic coherence across topics.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
topics_top_words	*topics_top_words*
corpus	*grouped_corpus* <*TokenizedDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
output	*named_file*
mean_coherence	*numeric_result*

Options

Name	Description	Type
coherence	Coherence measure to use, selecting from one of Gensim's pre-defined measures: 'u_mass', 'c_v', 'c_uci', 'c_npmi'. Default: 'u_mass'	'u_mass', 'c_v', 'c_uci' or 'c_npmi'
window_size	Size of the window to be used for coherence measures using boolean sliding window as their probability estimator. For 'u_mass' this doesn't matter. If None, the default window sizes are used which are: 'c_v' - 110, 'c_uci' - 10, 'c_npmi' - 10.	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_topic_coherence_module]
type=pimlico.modules.gensim.coherence
input_topics_top_words=module_a.some_output
input_corpus=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_topic_coherence_module]
type=pimlico.modules.gensim.coherence
input_topics_top_words=module_a.some_output
input_corpus=module_a.some_output
input_vocab=module_a.some_output
coherence=u_mass
window_size=0
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *lda_coherence*

LDA trainer

Path	pimlico.modules.gensim.lda
Executable	yes

Trains LDA using Gensim's basic LDA implementation, or the multicore version.

Does not support Python 2, since Gensim has dropped Python 2 support.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*IntegerListsDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
model	*lda_model*

Options

Name	Description	Type
alpha	Alpha prior over topic distribution. May be one of special values 'symmetric', 'asymmetric' and 'auto', or a single float, or a list of floats. Default: symmetric	'symmetric', 'asymmetric', 'auto' or a float
chunksize	Model's chunksize parameter. Chunk size to use for distributed/multicore computing. Default: 2000	int
decay	Decay parameter. Default: 0.5	float
distributed	Turn on distributed computing. Default: False. Ignored by multicore implementation	bool
eta	Eta prior of word distribution. May be one of special values 'auto' and 'symmetric', or a float. Default: symmetric	'symmetric', 'auto' or a float
eval_every		int
gamma_threshold		float
ignore_terms	Ignore any of these terms in the bags of words when iterating over the corpus to train the model. Typically, you'll want to include an OOV term here if your corpus has one, and any other special terms that are not part of a document's content	comma-separated list of strings
iterations	Max number of iterations in each update. Default: 50	int
minimum_phi_value		float
minimum_probability		float
multicore	Use Gensim's multicore implementation of LDA training (gensim.models.ldamulticore). Default is to use gensim.models.ldamodel. Number of cores used for training set by Pimlico's processes parameter	bool
num_topics	Number of topics for the trained model to have. Default: 100	int
offset	Offset parameter. Default: 1.0	float
passes	Passes parameter. Default: 1	int
tfidf	Transform word counts using TF-IDF when presenting documents to the model for training. Default: False	bool
update_every	Model's update_every parameter. Default: 1. Ignored by multicore implementation	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_lda_trainer_module]
type=pimlico.modules.gensim.lda
input_corpus=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_lda_trainer_module]
type=pimlico.modules.gensim.lda
input_corpus=module_a.some_output
input_vocab=module_a.some_output
alpha=symmetric
```

(continues on next page)

(continued from previous page)

```
chunksize=2000
decay=0.50
distributed=F
eta=symmetric
eval_every=10
gamma_threshold=0.00
ignore_terms=
iterations=50
minimum_phi_value=0.01
minimum_probability=0.01
multicore=F
num_topics=100
offset=1.00
passes=1
tfidf=F
update_every=1
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *lda_train*

LDA document topic analysis

Path	pimlico.modules.gensim.lda_doc_topics
Executable	yes

Takes a trained LDA model and produces the topic vector for every document in a corpus.

The corpus is given as integer lists documents, which are the integer IDs of the words in each sentence of each document. It is assumed that the corpus uses the same vocabulary to map to integer IDs as the LDA model's training corpus, so no further mapping needs to be done.

Does not support Python 2 since Gensim has dropped Python 2 support.

Todo: Add test pipeline and test

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*IntegerListsDocumentType*>
model	*lda_model*

Outputs

Name	Type(s)
vectors	*grouped_corpus* <*VectorDocumentType*>

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_lda_doc_topics_module]
type=pimlico.modules.gensim.lda_doc_topics
input_corpus=module_a.some_output
input_model=module_a.some_output
```

LDA top words

Path	pimlico.modules.gensim.lda_top_words
Executable	yes

Extract the top words for each topic from a Gensim LDA model.

Can be used as input to coherence evaluation.

Currently, this just outputs the highest probability words, but it could be extended in future to extract words according to other measures, like relevance or lift.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
model	*lda_model*

Outputs

Name	Type(s)
top_words	*topics_top_words*

Options

Name	Description	Type
num_words	Number of words to show per topic. Default: 15	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_lda_top_words_module]
type=pimlico.modules.gensim.lda_top_words
input_model=module_a.some_output
```

This example usage includes more options.

```
[my_lda_top_words_module]
type=pimlico.modules.gensim.lda_top_words
input_model=module_a.some_output
num_words=15
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *lda_top_words*

LDA-seq (DTM) trainer

Path	pimlico.modules.gensim.ldaseq
Executable	yes

Trains DTM using Gensim's DTM implementation.

Documents in the input corpus should be accompanied by an aligned corpus of string labels, where each time slice is represented by a label. The slices should be ordered, so all instances of a given label should be in sequence.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
corpus	*grouped_corpus*<*IntegerListsDocumentType*>
labels	*grouped_corpus*<*LabelDocumentType*>
vocab	*dictionary*

Outputs

Name	Type(s)
model	ldaseq_model

Options

Name	Description	Type
alphas	The prior probability for the model	float
chain_variance	Gaussian parameter defined in the beta distribution to dictate how the beta values evolve over time.	float
chunksize	Model's chunksize parameter. Chunk size to use for distributed/multicore computing. Default: 2000.	int
em_max_iter	Maximum number of iterations until converge of the Expectation-Maximization algorithm	int
em_min_iter	Minimum number of iterations until converge of the Expectation-Maximization algorithm	int
ignore_terms	Ignore any of these terms in the bags of words when iterating over the corpus to train the model. Typically, you'll want to include an OOV term here if your corpus has one, and any other special terms that are not part of a document's content	comma-separated list of strings
lda_inference_max_iter	Maximum number of iterations in the inference step of the LDA training. Default: 25	int
num_topics	Number of topics for the trained model to have. Default: 100	int
passes	Number of passes over the corpus for the initial LDA model. Default: 10	int
tfidf	Transform word counts using TF-IDF when presenting documents to the model for training. Default: False	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_ldaseq_trainer_module]
type=pimlico.modules.gensim.ldaseq
input_corpus=module_a.some_output
input_labels=module_a.some_output
input_vocab=module_a.some_output
```

This example usage includes more options.

```
[my_ldaseq_trainer_module]
type=pimlico.modules.gensim.ldaseq
input_corpus=module_a.some_output
input_labels=module_a.some_output
input_vocab=module_a.some_output
alphas=0.01
chain_variance=0.01
chunksize=100
em_max_iter=20
em_min_iter=6
ignore_terms=
```

(continues on next page)

(continued from previous page)

```
lda_inference_max_iter=25
num_topics=100
passes=10
tfidf=F
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *dtm_train*

LDA-seq (DTM) document topic analysis

Path	pimlico.modules.gensim.ldaseq_doc_topics
Executable	yes

Takes a trained DTM model and produces the topic vector for every document in a corpus.

The corpus is given as integer lists documents, which are the integer IDs of the words in each sentence of each document. It is assumed that the corpus uses the same vocabulary to map to integer IDs as the LDA model's training corpus, so no further mapping needs to be done.

We also require a corpus of labels to say what time slice each document is in. These should be from the same set of labels that the DTM model was trained on, so that each document label can be mapped to a trained slice.

Does not support Python 2 since Gensim has dropped Python 2 support.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*IntegerListsDocumentType*>
labels	*grouped_corpus* <*LabelDocumentType*>
model	ldaseq_model

Outputs

Name	Type(s)
vectors	*grouped_corpus* <*VectorDocumentType*>

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_ldaseq_doc_topics_module]
type=pimlico.modules.gensim.ldaseq_doc_topics
input_corpus=module_a.some_output
input_labels=module_a.some_output
input_model=module_a.some_output
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *dtm_infer*

1.3.4 Input readers

Various input readers for various datatypes. These are used to read in data from some external source, such as a corpus in its distributed format (e.g. XML files or a collection of text files), and present it to the Pimlico pipeline as a Pimlico dataset, which can be used as input to other modules.

They do not typically store the data as a Pimlico dataset, but produce it on the fly, although sometimes it could be appropriate to do otherwise.

Note that there can be multiple input readers for a single datatype. For example, there are many ways to read in a corpus of raw text documents, depending on the format they're stored in. They might by in one big XML file, text files collected into compressed archives, a big text file with document separators, etc. These all require their own input reader and all of them produce the same output corpus type.

Embeddings

Read vector embeddings (e.g. word embeddings) from various storage formats.

There are several formats in common usage and we provide readers for most of these here: *FastText*, *word2vec* and *GloVe*.

FastText embedding reader (bin)

Path	pimlico.modules.input.embeddings.fasttext
Executable	yes

Reads in embeddings from the FastText format, storing them in the format used internally in Pimlico for embeddings.

Loads the fastText `.bin` format using the fasttext library itself. Outputs both a fixed set of embeddings in Pimlico's standard format and a special fastText datatype that provides access to more features of the model.

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*embeddings*
model	fasttext_embeddings

Options

Name	Description	Type
path	(required) Path to the FastText embedding file	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fasttext_bin_embedding_reader_module]
type=pimlico.modules.input.embeddings.fasttext
path=value
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *fasttext_input_test*

FastText embedding reader (Gensim)

Path	pimlico.modules.input.embeddings.fasttext_gensim
Executable	yes

Reads in embeddings from the FastText format, storing them in the format used internally in Pimlico for embeddings. This version uses Gensim's implementation of the format reader, so depends on Gensim.

Can be used, for example, to read the pre-trained embeddings offered by Facebook AI.

Reads only the binary format (`.bin`), not the text format (`.vec`).

Does not support Python 2, since Gensim has dropped Python 2 support.

See also:

***pimlico.modules.input.embeddings.fasttext*:** An alternative reader that does not use Gensim. It permits (only) reading the text format.

Todo: Add test pipeline. This is slightly difficult, as we need a small FastText binary file, which is harder to produce, since you can't easily just truncate a big file.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*embeddings*

Options

Name	Description	Type
path	(required) Path to the FastText embedding file (.bin)	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fasttext_embedding_reader_gensim_module]
type=pimlico.modules.input.embeddings.fasttext_gensim
path=value
```

FastText embedding reader (vec)

Path	pimlico.modules.input.embeddings.fasttext_vec
Executable	yes

Reads in embeddings from the FastText format, storing them in the format used internally in Pimlico for embeddings.

Can be used, for example, to read the pre-trained embeddings offered by Facebook AI.

Currently only reads the text format (`.vec`), not the binary format (`.bin`).

See also:

***pimlico.modules.input.embeddings.fasttext_gensim*:** An alternative reader that uses Gensim's FastText format reading code and permits reading from the binary format, which contains more information.

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*embeddings*

Options

Name	Description	Type
limit	Limit to the first N words. Since the files are typically ordered from most to least frequent, this limits to the N most common words	int
path	(required) Path to the FastText embedding file	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_fasttext_vec_embedding_reader_module]
type=pimlico.modules.input.embeddings.fasttext_vec
path=value
```

This example usage includes more options.

```
[my_fasttext_vec_embedding_reader_module]
type=pimlico.modules.input.embeddings.fasttext_vec
limit=0
path=value
```

GloVe embedding reader (Gensim)

Path	pimlico.modules.input.embeddings.glove
Executable	yes

Reads in embeddings from the GloVe format, storing them in the format used internally in Pimlico for embeddings. We use Gensim's implementation of the format reader, so the module depends on Gensim.

Can be used, for example, to read the pre-trained embeddings offered by Stanford.

Note that the format is almost identical to *word2vec*'s text format.

Note that this requires a recent version of Gensim, since they changed their KeyedVectors data structure. This is not enforced by the dependency check, since we're not able to require a specific version yet.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*embeddings*

Options

Name	Description	Type
path	(required) Path to the GloVe embedding file	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_glove_embedding_reader_module]
type=pimlico.modules.input.embeddings.glove
path=value
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *glove_input_test*

Word2vec embedding reader (Gensim)

Path	pimlico.modules.input.embeddings.word2vec
Executable	yes

Reads in embeddings from the word2vec format, storing them in the format used internally in Pimlico for embeddings. We use Gensim's implementation of the format reader, so the module depends on Gensim.

Can be used, for example, to read the pre-trained embeddings offered by Google.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*embeddings*

Options

Name	Description	Type
binary	Assume input is in word2vec binary format. Default: True	bool
limit	Limit to the first N vectors in the file. Default: no limit	int
path	(required) Path to the word2vec embedding file (.bin)	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_word2vec_embedding_reader_module]
type=pimlico.modules.input.embeddings.word2vec
path=value
```

This example usage includes more options.

```
[my_word2vec_embedding_reader_module]
type=pimlico.modules.input.embeddings.word2vec
binary=T
limit=0
path=value
```

Text corpora

20 Newsgroups

20 Newsgroups fetcher (sklearn)

Path	pimlico.modules.input.text.20newsgroups.sklearn_download
Executable	yes

Input reader to fetch the 20 Newsgroups dataset from Sklearn. See: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html

The original data can be downloaded from http://qwone.com/~jason/20Newsgroups/.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

No inputs

Outputs

Name	Type(s)
text	*grouped_corpus*<*RawTextDocumentType*>
labels	*grouped_corpus*<*IntegerDocumentType*>

Options

Name	Description	Type
limit	Truncate corpus	int
random_state	Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple runs	int
remove	May contain any subset of ('headers', 'footers', 'quotes'). Each of these are kinds of text that will be detected and removed from the newsgroup posts, preventing classifiers from overfitting on metadata	comma-separated list of strings
shuffle	Whether or not to shuffle the data: might be important for models that make the assumption that the samples are independent and identically distributed (i.i.d.), such as stochastic gradient descent	bool
subset	Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering	'train', 'test' or 'all'

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_20ng_fetcher_module]
type=pimlico.modules.input.text.20newsgroups.sklearn_download
```

This example usage includes more options.

```
[my_20ng_fetcher_module]
type=pimlico.modules.input.text.20newsgroups.sklearn_download
limit=0
random_state=0
remove=text,text,...
shuffle=T
subset=train
```

Europarl corpus reader

Path	pimlico.modules.input.text.europarl
Executable	no

Input reader for raw, unaligned text from Europarl corpus. This does not cover the automatically aligned versions of the corpus that are typically used for Machine Translation.

The module takes care of a bit of extra processing specific to cleaning up the Europarl data.

See also:

raw_text_files, which this extends with special postprocessing.

This is an input module. It takes no pipeline inputs and is used to read in data

Inputs

No inputs

Outputs

Name	Type(s)
corpus	*grouped_corpus* <*RawTextDocumentType*>

Options

Name	Description	Type
archive_basename	Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Number of documents to include in each archive (default: 1k)	int
encoding	Encoding to assume for input files. Default: utf8	string
encoding_errors	What to do in the case of invalid characters in the input while decoding (e.g. illegal utf-8 chars). Select 'strict' (default), 'ignore', 'replace'. See Python's str.decode() for details	string
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	comma-separated list of strings
files	(required) Comma-separated list of absolute paths to files to include in the collection. Paths may include globs. Place a '?' at the start of a filename to indicate that it's optional. You can specify a line range for the file by adding ':X-Y' to the end of the path, where X is the first line and Y the last to be included. Either X or Y may be left empty. (Line numbers are 1-indexed.)	comma-separated list of (line range-limited) file paths

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_europarl_reader_module]
type=pimlico.modules.input.text.europarl
files=path1,path2,...
```

This example usage includes more options.

```
[my_europarl_reader_module]
type=pimlico.modules.input.text.europarl
archive_basename=archive
```

(continues on next page)

(continued from previous page)

```
archive_size=1000
encoding=utf8
encoding_errors=strict
exclude=text,text,...
files=path1,path2,...
```

Huggingface text corpus

Path	pimlico.modules.input.text.huggingface
Executable	yes

Input reader to fetch a text corpus from Huggingface's datasets library. See: https://huggingface.co/datasets/.

Uses Huggingface's `load_dataset()` function to download a dataset and then converts it to a Pimlico raw text archive.

This module does not support Python 2, so can only be used when Pimlico is being run under Python 3

Inputs

No inputs

Outputs

Name	Type(s)
default	*grouped_corpus* <*RawTextDocumentType*>

Further conditional outputs

In addition to the default output `default`, if more than one column is specified, further outputs will be provided, each containing a column and named after the column.

The first column name given is always provided as the first (default) output, called "default".

Options

Name	Description	Type
columns	(required) Name(s) of column(s) to store as Pimlico datasets. At least one must be given	comma-separated list of strings
dataset	(required) Name of the dataset to download	string
doc_name	Take the doc names from the named column. The special value 'enum' (default) just numbers the sequence of documents	string
name	Name defining the dataset configuration. This corresponds to the second argument of load_dataset()	string
split	Restrict to a split of the data. Must be one of the splits that this dataset provides. The default value of 'train' will work for many datasets, but is not guaranteed to be appropriate	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_huggingface_text_module]
type=pimlico.modules.input.text.huggingface
columns=text,text,...
dataset=value
```

This example usage includes more options.

```
[my_huggingface_text_module]
type=pimlico.modules.input.text.huggingface
columns=text,text,...
dataset=value
doc_name=enum
name=value
split=train
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *huggingface_dataset*

Raw text archives

Path	pimlico.modules.input.text.raw_text_archives
Executable	yes

Input reader for raw text file collections stored in archives. Reads archive files from arbitrary locations specified by a list of and iterates over the files they contain.

The input paths must be absolute paths, but remember that you can make use of various *special substitutions in the config file* to give paths relative to your project root, or other locations.

Unlike *raw_text_files*, globs are not permitted. There's no reason why they could not be, but they are not allowed for now, to keep these modules simpler. This feature could be added, or if you need it, you could create your own input reader module based on this one.

All paths given are assumed to be required for the dataset to be ready, unless they are preceded by a `?`.

It can take a long time to count up the files in an archive, if there are a lot of them, as we need to iterate over the whole archive. If a file is found with a path and name identical to the tar archive's, with the suffix `.count`, a document count will be read from there and used instead of counting. Make sure it is correct, as it will be blindly trusted, which will cause difficulties in your pipeline if it's wrong! The file is expected to contain a single integer as text.

All files in the archive are included. If you wish to filter files or preprocess them somehow, this can be easily done by subclassing `RawTextArchivesInputReader` and overriding appropriate bits, e.g. *RawTextArchivesInputReader.Setup.iter_archive_infos()*. You can then use this reader to create an input reader module with the factory function, as is done here.

See also:

raw_text_files for raw files not in archives

This is an input module. It takes no pipeline inputs and is used to read in data

Inputs

No inputs

Outputs

Name	Type(s)
corpus	*grouped_corpus* <*RawTextDocumentType*>

Options

Name	Description	Type
archive_basename	Basename to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Number of documents to include in each archive (default: 1k)	int
encoding	Encoding to assume for input files. Default: utf8	string
encoding_errors	What to do in the case of invalid characters in the input while decoding (e.g. illegal utf-8 chars). Select 'strict' (default), 'ignore', 'replace'. See Python's str.decode() for details	string
files	(required) Comma-separated list of absolute paths to files to include in the collection. Place a '?' at the start of a filename to indicate that it's optional	absolute file path

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_raw_text_archives_reader_module]
type=pimlico.modules.input.text.raw_text_archives
files=path1,path2,...
```

This example usage includes more options.

```
[my_raw_text_archives_reader_module]
type=pimlico.modules.input.text.raw_text_archives
archive_basename=archive
archive_size=1000
encoding=utf8
encoding_errors=strict
files=path1,path2,...
```

Raw text files

Path	pimlico.modules.input.text.raw_text_files
Executable	no

Input reader for raw text file collections. Reads in files from arbitrary locations specified by a list of globs.

The input paths must be absolute paths (or globs), but remember that you can make use of various *special substitutions in the config file* to give paths relative to your project root, or other locations.

The file paths may use globs to match multiple files. By default, it is assumed that every filename should exist and every glob should match at least one file. If this does not hold, the dataset is assumed to be not ready. You can override this by placing a `?` at the start of a filename/glob, indicating that it will be included if it exists, but is not depended on for considering the data ready to use.

This is an input module. It takes no pipeline inputs and is used to read in data

Inputs

No inputs

Outputs

Name	Type(s)
corpus	*grouped_corpus* <*RawTextDocumentType*>

Options

Name	Description	Type
archive_basename	Base name to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Number of documents to include in each archive (default: 1k)	int
encoding	Encoding to assume for input files. Default: utf8	string
encoding_errors	What to do in the case of invalid characters in the input while decoding (e.g. illegal utf-8 chars). Select 'strict' (default), 'ignore', 'replace'. See Python's str.decode() for details	string
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	absolute file path
files	(required) Comma-separated list of absolute paths to files to include in the collection. Paths may include globs. Place a '?' at the start of a filename to indicate that it's optional. You can specify a line range for the file by adding ':X-Y' to the end of the path, where X is the first line and Y the last to be included. Either X or Y may be left empty. (Line numbers are 1-indexed.)	comma-separated list of (line range-limited) file paths

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_raw_text_files_reader_module]
type=pimlico.modules.input.text.raw_text_files
files=path1,path2,...
```

This example usage includes more options.

```
[my_raw_text_files_reader_module]
type=pimlico.modules.input.text.raw_text_files
archive_basename=archive
archive_size=1000
encoding=utf8
encoding_errors=strict
exclude=path1,path2,...
files=path1,path2,...
```

XML files

Path	pimlico.modules.input.xml
Executable	yes

Input reader for XML file collections. Gigaword, for example, is stored in this way. The data retrieved from the files is plain unicode text.

Todo: Add test pipeline

This is an input module. It takes no pipeline inputs and is used to read in data

Inputs

No inputs

Outputs

Name	Type(s)
corpus	*grouped_corpus*<*RawTextDocumentType*>

Options

Name	Description	Type
archive_basename	Basename to use for archive tar files. The archive number is appended to this. (Default: 'archive')	string
archive_size	Number of documents to include in each archive (default: 1k)	int
document_name_attr	Attribute of document nodes to get document name from. Use special value 'filename' to use the filename (without extensions) as a document name. In this case, if there's more than one doc in a file, an integer is appended to the doc name after the first doc. (Default: 'filename')	string
document_node_type	XML node type to extract documents from (default: 'doc')	string
encoding	Encoding to assume for input files. Default: utf8	string
encoding_errors	What to do in the case of invalid characters in the input while decoding (e.g. illegal utf-8 chars). Select 'strict' (default), 'ignore', 'replace'. See Python's str.decode() for details	string
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	absolute file path
files	(required) Comma-separated list of absolute paths to files to include in the collection. Paths may include globs. Place a '?' at the start of a filename to indicate that it's optional	absolute file path
filter_on_doc_attr	Comma-separated list of key=value constraints. If given, only docs with the attribute 'key' on their doc node and the attribute value 'value' will be included	comma-separated list of key=value constraints

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_xml_files_reader_module]
type=pimlico.modules.input.xml
files=path1,path2,...
```

This example usage includes more options.

```
[my_xml_files_reader_module]
type=pimlico.modules.input.xml
archive_basename=archive
archive_size=1000
document_name_attr=filename
document_node_type=doc
encoding=utf8
encoding_errors=strict
exclude=path1,path2,...
files=path1,path2,...
filter_on_doc_attr=value
```

1.3.5 Malt dependency parser

Path	pimlico.modules.malt
Executable	yes

Runs the Malt dependency parser.

Malt is a Java tool, so we use a Py4J wrapper.

Input is supplied as word annotations (which are converted to CoNLL format for input to the parser). These must include at least each word (field 'word') and its POS tag (field 'pos'). If a 'lemma' field is supplied, that will also be used.

The fields in the output contain all of the word features provided by the parser's output. Some may be `None` if they are empty in the parser output. All the fields in the input (which always include `word` and `pos` at least) are also output.

Inputs

Name	Type(s)
documents	*grouped_corpus* <*WordAnnotationsDocumentType*>

Outputs

Name	Type(s)
parsed	*AddAnnotationField*

Options

Name	Description	Type
model	Filename of parsing model, or path to the file. If just a filename, assumed to be Malt models dir (models/malt). Default: engmalt.linear-1.7.mco, which can be acquired by 'make malt' in the models dir	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_malt_module]
type=pimlico.modules.malt
input_documents=module_a.some_output
```

This example usage includes more options.

```
[my_malt_module]
type=pimlico.modules.malt
input_documents=module_a.some_output
model=engmalt.linear-1.7.mco
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *malt_parse*

1.3.6 NLTK

Modules that wrap functionality in the Natural Language Toolkit (NLTK).

Currently, not much is provided here, but adding new modules is easy to do, so hopefully more modules will gradually appear.

NIST tokenizer

Path	pimlico.modules.nltk.nist_tokenize
Executable	yes

Sentence splitting and tokenization using the NLTK NIST tokenizer.

Very simple tokenizer that's fairly language-independent and doesn't need a trained model. Use this if you just need a rudimentary tokenization (though more sophisticated than *simple_tokenize*).

Inputs

Name	Type(s)
text	*grouped_corpus*<*RawTextDocumentType*>

Outputs

Name	Type(s)
documents	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
lowercase	Lowercase all output. Default: False	bool
non_european	Use the tokenizer's international_tokenize() method instead of tokenize(). Default: False	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_nltk_nist_tokenizer_module]
type=pimlico.modules.nltk.nist_tokenize
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_nltk_nist_tokenizer_module]
type=pimlico.modules.nltk.nist_tokenize
input_text=module_a.some_output
lowercase=F
non_european=F
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *nltk_nist_tokenize*
- *nltk_nist_tokenize*

1.3.7 OpenNLP tools

A collection of module types to wrap individual OpenNLP tools.

At the moment, this includes several tool. A few other modules have been here previously, but have not yet been updated to the new datatypes system. See `pimlico.old_datatypes.modules.opennlp`.

Other OpenNLP tools can be wrapped fairly straightforwardly following the same pattern, using Py4J.

Constituency parser

Path	pimlico.modules.opennlp.parse
Executable	yes

Constituency parsing using OpenNLP's tools.

We run OpenNLP in the background using a Py4J wrapper, just as with the other OpenNLP wrappers.

The output format is not yet ideal: currently we produce documents consisting of a list of strings, each giving the OpenNLP tree output for a sentence. It would be better to use a standard constituency tree datatype that can be used generically as input to any modules required tree input. For now, if you write a module taking input from the parser, it will itself need to process the strings from the OpenNLP parser output.

Inputs

Name	Type(s)
documents	*grouped_corpus* <*TokenizedDocumentType*>

Outputs

Name	Type(s)
trees	*grouped_corpus* <*OpenNLPTreeStringsDocumentType*>

Options

Name	Description	Type
model	Parser model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/)	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_opennlp_parser_module]
type=pimlico.modules.opennlp.parse
input_documents=module_a.some_output
```

This example usage includes more options.

```
[my_opennlp_parser_module]
type=pimlico.modules.opennlp.parse
input_documents=module_a.some_output
model=en-parser-chunking.bin
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *opennlp_parse*

POS-tagger

Path	pimlico.modules.opennlp.pos
Executable	yes

Part-of-speech tagging using OpenNLP's tools.

By default, uses the pre-trained English model distributed with OpenNLP. If you want to use other models (e.g. for other languages), download them from the OpenNLP website to the models dir (*models/opennlp*) and specify the model name as an option.

Inputs

Name	Type(s)
text	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
pos	*grouped_corpus*<*WordAnnotationsDocumentType*>

Options

Name	Description	Type
model	POS tagger model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_opennlp_pos_tagger_module]
type=pimlico.modules.opennlp.pos
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_opennlp_pos_tagger_module]
type=pimlico.modules.opennlp.pos
input_text=module_a.some_output
model=en-pos-maxent.bin
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *opennlp_pos*

Tokenizer

Path	pimlico.modules.opennlp.tokenize
Executable	yes

Sentence splitting and tokenization using OpenNLP's tools.

Sentence splitting may be skipped by setting the option *tokenize_only=T*. The tokenizer will then assume that each line in the input file represents a sentence and tokenize within the lines.

Inputs

Name	Type(s)
text	*grouped_corpus*<*TextDocumentType*>

Outputs

Name	Type(s)
documents	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
sentence_model	Sentence segmentation model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string
token_model	Tokenization model. Specify a full path, or just a filename. If a filename is given it is expected to be in the opennlp model directory (models/opennlp/)	string
tokenize_only	By default, sentence splitting is performed prior to tokenization. If tokenize_only is set, only the tokenization step is executed	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_opennlp_tokenizer_module]
type=pimlico.modules.opennlp.tokenize
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_opennlp_tokenizer_module]
type=pimlico.modules.opennlp.tokenize
input_text=module_a.some_output
sentence_model=en-sent.bin
token_model=en-token.bin
tokenize_only=F
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *opennlp_tokenize*

1.3.8 Output modules

Modules that only have inputs and write output to somewhere outside the Pimlico pipeline.

Text corpus directory

Path	pimlico.modules.output.text_corpus
Executable	yes

Output module for producing a directory containing a text corpus, with documents stored in separate files.

The input must be a raw text grouped corpus. Corpora with other document types can be converted to raw text using the *format* module.

Inputs

Name	Type(s)
corpus	*grouped_corpus*<*RawTextDocumentType*>

Outputs

No outputs

Options

Name	Description	Type
archive_dirs	Create a subdirectory for each archive of the grouped corpus to store that archive's documents in. Otherwise, all documents are stored in the same directory (or subdirectories where the document names include directory separators)	bool
invalid	What to do with invalid documents (where there's been a problem reading/processing the document somewhere in the pipeline). 'skip' (default): don't output the document at all. 'empty': output an empty file	'skip' or 'empty'
path	(required) Directory to write the corpus to	string
suffix	Suffix to use for each document's filename	string
tar	Add all files to a single tar archive, instead of just outputting to disk in the given directory. This is a good choice for very large corpora, for which storing to files on disk can cause filesystem problems. If given, the value is used as the basename for the tar archive. Default: do not output tar	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_text_corpus_module]
type=pimlico.modules.output.text_corpus
input_corpus=module_a.some_output
path=value
```

This example usage includes more options.

```
[my_text_corpus_module]
type=pimlico.modules.output.text_corpus
input_corpus=module_a.some_output
archive_dirs=T
invalid=skip
path=value
suffix=value
tar=value
```

1.3.9 Scikit-learn tools

Scikit-learn ('sklearn') provides easy-to-use implementations of a large number of machine-learning methods, based on Numpy/Scipy.

You can build Numpy arrays from your corpus using the `feature processing tools` and then use them as input to Scikit-learn's tools using the modules in this package.

Sklearn logistic regression

Path	pimlico.modules.sklearn.logistic_regression
Executable	yes

Provides an interface to Scikit-Learn's simple logistic regression trainer.

You may also want to consider using:

- LogisticRegressionCV: LR with cross-validation to choose regularization strength
- SGDClassifier: general gradient-descent training for classifiers, which includes logistic regression. A better choice for training on a large dataset.

Inputs

Name	Type(s)
features	*scored_real_feature_sets*

Outputs

Name	Type(s)
model	*sklearn_model*

Options

Name	Description	Type
options	Options to pass into the constructor of LogisticRegression, formatted as a JSON dictionary (potentially without the {}s). E.g.: '"C":1.5, "penalty":"l2"'	JSON dict

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_sklearn_log_reg_module]
type=pimlico.modules.sklearn.logistic_regression
input_features=module_a.some_output
```

This example usage includes more options.

```
[my_sklearn_log_reg_module]
type=pimlico.modules.sklearn.logistic_regression
input_features=module_a.some_output
options="C":1.5, "penalty":"l2"
```

1.3.10 spaCy

Run spaCy tools and pipelines on your datasets.

Currently only includes tokenization, but this could be expanded to include many more of spaCy's tools.

Or, if you want a different tool/pipeline, you could create your own module type following the same approach.

NP chunk extractor

Path	pimlico.modules.spacy.extract_nps
Executable	yes

Extract NP chunks

Performs the full spaCy pipeline including tokenization, sentence segmentation, POS tagging and parsing and outputs documents containing only a list of the noun phrase chunks that were found by the parser.

This functionality is provided very conveniently by spaCy's `Doc.noun_chunks` after parsing, so this is a light wrapper around spaCy.

The output is presented as a tokenized document. Each sentence in the document represents a single NP.

Inputs

Name	Type(s)
text	*grouped_corpus*<*RawTextDocumentType*>

Outputs

Name	Type(s)
nps	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
model	spaCy model to use. This may be a name of a standard spaCy model or a path to the location of a trained model on disk, if on_disk=T. If it's not a path, the spaCy download command will be run before execution	string
on_disk	Load the specified model from a location on disk (the model parameter gives the path)	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_spacy_extract_nps_module]
type=pimlico.modules.spacy.extract_nps
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_spacy_extract_nps_module]
type=pimlico.modules.spacy.extract_nps
input_text=module_a.some_output
model=en_core_web_sm
on_disk=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *spacy_parse_text*

Text parser

Path	pimlico.modules.spacy.parse_text
Executable	yes

Parsing using spaCy

Entire parsing pipeline from raw text using the same spaCy model.

The word annotations in the output contain the information from the spaCy parser and the documents are split into sentences following the spaCy's sentence segmentation.

The annotation fields follow those produced by the Malt parser: pos, head and deprel.

Inputs

Name	Type(s)
text	*grouped_corpus* <*RawTextDocumentType*>

Outputs

Name	Type(s)
parsed	*grouped_corpus* <*WordAnnotationsDocumentType*>

Options

Name	Description	Type
model	spaCy model to use. This may be a name of a standard spaCy model or a path to the location of a trained model on disk, if on_disk=T. If it's not a path, the spaCy download command will be run before execution	string
on_disk	Load the specified model from a location on disk (the model parameter gives the path)	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_spacy_text_parser_module]
type=pimlico.modules.spacy.parse_text
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_spacy_text_parser_module]
type=pimlico.modules.spacy.parse_text
input_text=module_a.some_output
model=en_core_web_sm
on_disk=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *spacy_parse_text*

Tokenizer

Path	pimlico.modules.spacy.tokenize
Executable	yes

Tokenization using spaCy.

Inputs

Name	Type(s)
text	*grouped_corpus* <*TextDocumentType*>

Outputs

Name	Type(s)
documents	*grouped_corpus* <*TokenizedDocumentType*>

Options

Name	Description	Type
model	spaCy model to use. This may be a name of a standard spaCy model or a path to the location of a trained model on disk, if on_disk=T. If it's not a path, the spaCy download command will be run before execution	string
on_disk	Load the specified model from a location on disk (the model parameter gives the path)	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_spacy_tokenizer_module]
type=pimlico.modules.spacy.tokenize
input_text=module_a.some_output
```

This example usage includes more options.

```
[my_spacy_tokenizer_module]
type=pimlico.modules.spacy.tokenize
input_text=module_a.some_output
model=en_core_web_sm
on_disk=T
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*
- *custom_module_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *spacy_tokenize*

1.3.11 Document-level text filters

Simple text filters that are applied at the document level, i.e. each document in a TarredCorpus is processed one at a time. These perform relatively simple processing, not relying on external software or involving lengthy processing times. They are therefore most often used using the `filter=T` option, so that the processing is performed on the fly.

Such filters are needed sometimes just to convert before different datapoint formats.

Probably a good deal of these will be added in due course.

Text to character level

Path	pimlico.modules.text.char_tokenize
Executable	yes

Filter to treat text data as character-level tokenized data. This makes it simple to train character-level models, since the output appears exactly like a tokenized document, where each token is a single character. You can then feed it into any module that expects tokenized text.

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*TextDocumentType*>

Outputs

Name	Type(s)
corpus	*grouped_corpus* <*CharacterTokenizedDocumentType*>

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_char_tokenize_module]
type=pimlico.modules.text.char_tokenize
input_corpus=module_a.some_output
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *simple_tokenize*

Normalize tokenized text

Path	pimlico.modules.text.normalize
Executable	yes

Perform text normalization on tokenized documents.

Currently, this includes the following:

- case normalization (to upper or lower case)
- blank line removal
- empty sentence removal
- punctuation removal
- removal of words that contain only punctuation
- numerical character removal
- minimum word length filter

In the future, more normalization operations may be added.

Inputs

Name	Type(s)
corpus	*grouped_corpus*<*TokenizedDocumentType*>

Outputs

Name	Type(s)
corpus	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
case	Transform all text to upper or lower case. Choose from 'upper' or 'lower', or leave blank to not perform transformation	'upper', 'lower' or ''
min_word_length	Remove any words shorter than this. Default: 0 (don't do anything)	int
remove_empty	Skip over any empty sentences (i.e. blank lines). Applied after other processing, so this will remove sentences that are left empty by other filters	bool
remove_nums	Remove numeric characters	bool
remove_only_punct	Skip over any sentences that are empty if punctuation is ignored	bool
remove_punct	Remove punctuation from all tokens and then remove the whole token if nothing's left	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_normalize_module]
type=pimlico.modules.text.normalize
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_normalize_module]
type=pimlico.modules.text.normalize
input_corpus=module_a.some_output
case=
min_word_length=0
remove_empty=F
remove_nums=F
remove_only_punct=F
remove_punct=F
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *train_tms_example*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *normalize*

Simple tokenization

Path	pimlico.modules.text.simple_tokenize
Executable	yes

Tokenize raw text using simple splitting.

This is useful where either you don't mind about the quality of the tokenization and just want to test something quickly, or text is actually already tokenized, but stored as a raw text datatype.

If you want to do proper tokenization, consider either the CoreNLP or OpenNLP core modules.

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*TextDocumentType*>

Outputs

Name	Type(s)
corpus	*grouped_corpus*<*TokenizedDocumentType*>

Options

Name	Description	Type
splitter	Character or string to split on. Default: space	string

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_simple_tokenize_module]
type=pimlico.modules.text.simple_tokenize
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_simple_tokenize_module]
type=pimlico.modules.text.simple_tokenize
input_corpus=module_a.some_output
splitter=
```

Example pipelines

This module is used by the following *example pipelines*. They are examples of how the module can be used together with other modules in a larger pipeline.

- *tokenize_example*
- *tokenize_example2*

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *simple_tokenize*

Normalize raw text

Path	pimlico.modules.text.text_normalize
Executable	yes

Text normalization for raw text documents.

Similar to *normalize* module, but operates on raw text, not pre-tokenized text, so provides a slightly different set of tools.

Inputs

Name	Type(s)
corpus	*grouped_corpus* <*TextDocumentType*>

Outputs

Name	Type(s)
corpus	*grouped_corpus* <*RawTextDocumentType*>

Options

Name	Description	Type
blank_lines	Remove all blank lines (after whitespace stripping, if requested)	bool
case	Transform all text to upper or lower case. Choose from 'upper' or 'lower', or leave blank to not perform transformation	'upper', 'lower' or ''
strip	Strip whitespace from the start and end of lines	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_text_normalize_module]
type=pimlico.modules.text.text_normalize
input_corpus=module_a.some_output
```

This example usage includes more options.

```
[my_text_normalize_module]
type=pimlico.modules.text.text_normalize
input_corpus=module_a.some_output
blank_lines=T
case=
strip=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *normalize*

1.3.12 General utilities

General utilities for things like filesystem manipulation.

Module output alias

Path	pimlico.modules.utility.alias
Executable	no

Alias a datatype coming from the output of another module.

Used to assign a handy identifier to the output of a module, so that we can just refer to this alias module later in the pipeline and use its default output. This can help make for a more readable pipeline config.

For example, say we use *split* to split a dataset into two random subsets. The two splits can be accessed by referring to the two outputs of that module: *split_module.set1* and *split_module.set2*. However, it's easy to lose track of what these splits are supposed to be used for, so we might want to give them names:

```
[split_module]
type=pimlico.modules.corpora.split
set1_size=0.2

[test_set]
type=pimlico.modules.utility.alias
input=split_module.set1

[training_set]
type=pimlico.modules.utility.alias
input=split_module.set2

[training_routine]
type=...
input_corpus=training_set
```

Note the difference between using this module and using the special *alias* module type. The *alias* type creates an alias for a whole module, allowing you to refer to all of its outputs, inherit its settings, and anything else you could do with the original module name. This module, however, provides an alias for exactly one output of a module and generates a module instance of its own in the pipeline (albeit a filter module).

Todo: Add test pipeline

This is a filter module. It is not executable, so won't appear in a pipeline's list of modules that can be run. It produces its output for the next module on the fly when the next module needs it.

Inputs

Name	Type(s)
input	*base_datatype*

Outputs

Name	Type(s)
output	same as input corpus

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_alias_module]
type=pimlico.modules.utility.alias
input_input=module_a.some_output
```

Collect files

Path	pimlico.modules.utility.collect_files
Executable	yes

Collect files output from different modules.

A simple convenience module to make it easier to inspect output by putting it all in one place.

Files are either collected into subdirectories or renamed to avoid clashes.

Inputs

Name	Type(s)
files	*list* of *named_file_collection*

Outputs

Name	Type(s)
files	collected_named_file_collection

Options

Name	Description	Type
names	List of string identifiers to use to distinguish the files from different sources, either used as subdirectory names or filename prefixes. If not given, integer ids will be used instead	absolute file path
sub-dirs	Use subdirectories to collect the files from different sources, rather than renaming each file. By default, a prefix is added to the filenames	bool

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_collect_files_module]
type=pimlico.modules.utility.collect_files
input_files=module_a.some_output
```

This example usage includes more options.

```
[my_collect_files_module]
type=pimlico.modules.utility.collect_files
input_files=module_a.some_output
names=path1,path2,...
subdirs=T
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *collect_files*

1.3.13 Visualization tools

Modules for plotting and suchlike

Bar chart plotter

Path	pimlico.modules.visualization.bar_chart
Executable	yes

Simple plotting of a bar chart from numeric results data using Matplotlib.

Inputs

Name	Type(s)
results	*list* of *numeric_result*

Outputs

Name	Type(s)
plot	*named_file_collection*

Options

Name	Description	Type
colors	Pyplot colors to use for each series. If shorter than the number of inputs, cycles round. Specify according to pyplot docs: https://matplotlib.org/2.0.2/api/colors_api.html. E.g. use single-letter color names, HTML color codes or HTML color names	absolute file path
labels	If given, a list of labels corresponding to the inputs to use in plots. Otherwise, inputs are numbered and the labels provided in their label fields are used	absolute file path

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_bar_chart_module]
type=pimlico.modules.visualization.bar_chart
input_results=module_a.some_output
```

This example usage includes more options.

```
[my_bar_chart_module]
type=pimlico.modules.visualization.bar_chart
input_results=module_a.some_output
colors=r,g,b,y,c,m,k
labels=path1,path2,...
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *bar_chart*

Embedding space plotter

Path	pimlico.modules.visualization.embeddings_plot
Executable	yes

Plot vectors from embeddings, trained by some other module, in a 2D space using a MDS reduction and Matplotlib.

They might, for example, come from *pimlico.modules.embeddings.word2vec*. The embeddings are read in using Pimlico's generic word embedding storage type.

Uses scikit-learn to perform the MDS/TSNE reduction.

The module outputs a Python file for doing the plotting (`plot.py`) and a CSV file containing the vector data (`data.csv`) that is used as input to the plotting. The Python file is then run to produce (if it succeeds) an output PDF (`plot.pdf`).

The idea is that you can use these source files (`plot.py` and `data.csv`) as a template and adjust the plotting code to produce a perfect plot for inclusion in your paper, website, desktop wallpaper, etc.

Inputs

Name	Type(s)
vectors	*list* of *embeddings*

Outputs

Name	Type(s)
plot	*named_file_collection*

Options

Name	Description	Type
cmap	Mapping from word prefixes to matplotlib plotting colours. Every word beginning with the given prefix has the prefix removed and is plotted in the corresponding colour. Specify as a JSON dictionary mapping prefix strings to colour strings	JSON string
colors	List of colours to use for different embedding sets. Should be a list of matplotlib colour strings, one for each embedding set given in input_vectors	absolute file path
metric	Distance metric to use. Choose from 'cosine', 'euclidean', 'manhattan'. Default: 'cosine'	'cosine', 'euclidean' or 'manhattan'
reduction	Dimensionality reduction technique to use to project to 2D. Available: mds (Multi-dimensional Scaling), tsne (t-distributed Stochastic Neighbor Embedding). Default: mds	'mds' or 'tsne'
skip	Number of most frequent words to skip, taking the next most frequent after these. Default: 0	int
words	Number of most frequent words to plot. Default: 50	int

Example config

This is an example of how this module can be used in a pipeline config file.

```
[my_embeddings_plot_module]
type=pimlico.modules.visualization.embeddings_plot
input_vectors=module_a.some_output
```

This example usage includes more options.

```
[my_embeddings_plot_module]
type=pimlico.modules.visualization.embeddings_plot
input_vectors=module_a.some_output
cmap={"key1":"value"}
colors=path1,path2,...
metric=cosine
reduction=mds
skip=0
words=50
```

Test pipelines

This module is used by the following *test pipelines*. They are a further source of examples of the module's usage.

- *embeddings_plot*

1.4 Command-line interface

The main Pimlico command-line interface (usually accessed via *pimlico.sh* in your project root) provides subcommands to perform different operations. Call it like so, using one of the subcommands documented below to access particular functionality:

```
./pimlico.sh <config-file> [general options...] <subcommand> [subcommand args/options]
```

The commands you are likely to use most often are: *status*, *run*, *reset* and maybe *browse*.

For a reference for each command's options, see the command-line documentation: `./pimlico.sh --help`, for a general reference and `./pimlico.sh <config_file> <command> --help` for a specific subcommand's reference.

Below is a more detailed guide for each subcommand, including all of the documentation available via the command line.

browse	View the data output by a module
clean	Remove all module output directories that do not correspond to a module in the pipeline
deps	List information about software dependencies: whether they're available, versions, etc
dump	Dump the entire available output data from a given pipeline module to a tarball
email	Test email settings and try sending an email using them
fixlength	Check the length of written outputs and fix it if it's wrong
inputs	Show the (expected) locations of the inputs of a given module
install	Install missing module library dependencies
jupyter	Create and start a new Jupyter notebook for the pipeline
licenses	List information about licsenses of software dependencies
load	Load a module's output data from a tarball previously created by the dump command
movestores	Move data between stores
newmodule	Create a new module type
output	Show the location where the given module's output data will be (or has been) stored
python	Load the pipeline config and enter a Python interpreter with access to it in the environment
recover	Examine and fix a partially executed map module's output state after forcible termination
reset	Delete any output from the given module and restore it to unexecuted state
run	Execute an individual pipeline module, or a sequence
shell	Open a shell to give access to the data output by a module
status	Output a module execution schedule for the pipeline and execution status for every module
stores	List named Pimlico stores
tar2pimarc	Convert grouped corpora from the old tar-based storage format to pimarc
unlock	Forcibly remove an execution lock from a module
variants	List the available variants of a pipeline config
visualize	Comming soon... visualize the pipeline in a pretty way

1.4.1 status

Command-line tool subcommand

Output a module execution schedule for the pipeline and execution status for every module.

Usage:

```
pimlico.sh [...] status [module_name] [-h] [--all] [--alias] [--short] [--history] [--
↪deps-of DEPS_OF] [--no-color] [--no-sections] [--expand-all] [--expand [EXPAND␣
↪[EXPAND ...]]]
```

Positional arguments

Arg	Description
[module_name]	Optionally specify a module name (or number). More detailed status information will be outut for this module. Alternatively, use this arg to limit the modules whose status will be output to a range by specifying 'A...B', where A and B are module names or numbers

Options

Option	Description
--all, -a	Show all modules defined in the pipeline, not just those that can be executed
--alias	Include module aliases after modules in the output. By default, they are not shown
--short, -s	Use a brief format when showing the full pipeline's status. Only applies when module names are not specified. This is useful with very large pipelines, where you just want a compact overview of the status
--history, -i	When a module name is given, even more detailed output is given, including the full execution history of the module
--deps-of, -d	Restrict to showing only the named/numbered module and any that are (transitive) dependencies of it. That is, show the whole tree of modules that lead through the pipeline to the given module
--no-color, --nc	Don't include terminal color characters, even if the terminal appears to support them. This can be useful if the automatic detection of color terminals doesn't work and the status command displays lots of horrible escape characters
--no-sections, --ns	Don't show section headings, but just a list of all the modules
--expand-all, --xa	Show section headings, expanding all
--expand, -x	Expand this section number. May be used multiple times. Give a section number like '1.2.3'. To expand the full subtree, give '1.2.3.'

1.4.2 variants

Command-line tool subcommand

List the available variants of a pipeline config

See *Pipeline variants* for more details.

Usage:

```
pimlico.sh [...] variants [-h]
```

1.4.3 run

Command-line tool subcommand

Main command for executing Pimlico modules from the command line *run* command.

Usage:

```
pimlico.sh [...] run [modules [modules ...]] [-h] [--force-rerun] [--all-deps] [--
↪all] [--dry-run] [--step] [--preliminary] [--exit-on-error] [--email {modend,end}]␣
↪[--last-error]
```

(continues on next page)

(continued from previous page)

Positional arguments

Arg	Description
[modules [modules ...]]	The name (or number) of the module to run. To run a stage from a multi-stage module, use 'module:stage'. Use 'status' command to see available modules. Use 'module:?' or 'module:help' to list available stages. If not given, defaults to next incomplete module that has all its inputs ready. You may give multiple modules, in which case they will be executed in the order specified

Options

Option	Description
--force-rerun, -f	Force running the module(s), even if it's already been run to completion
--all-deps, -a	If the given module(s) has dependent modules that have not been completed, executed them first. This allows you to specify a module late in the pipeline and execute the full pipeline leading to that point
--all	Run all currently unexecuted modules that have their inputs ready, or will have by the time previous modules are run. (List of modules will be ignored)
--dry-run, --dry, --check	Perform all pre-execution checks, but don't actually run the module(s)
--step	Enabled super-verbose debugging mode, which steps through a module's processing outputting a lot of information and allowing you to control the output as it goes. Useful for working out what's going on inside a module if it's mysteriously not producing the output you expected
--preliminary, --pre	Perform a preliminary run of any modules that take multiple datasets into one of their inputs. This means that we will run the module even if not all the datasets are yet available (but at least one is) and mark it as preliminarily completed
--exit-on-error	If an error is encountered while executing a module that causes the whole module execution to fail, output the error and exit. By default, Pimlico will send error output to a file (or print it in debug mode) and continue to execute the next module that can be executed, if any
--email	Send email notifications when processing is complete, including information about the outcome. Choose from: 'modend' (send notification after module execution if it fails and a summary at the end of everything), 'end' (send only the final summary). Email sending must be configured: see 'email' command to test
--last-error, -e	Don't execute, just output the error log from the last execution of the given module(s)

1.4.4 recover

Command-line tool subcommand

When a document map module gets killed forcibly, sometimes it doesn't have time to save its execution state, meaning that it can't pick up from where it left off.

Todo: This has not been updated for the Pimarc internal storage format, so still assumes that tar files are used. It will

be updated in future, if there is a need for it.

This command tries to fix the state so that execution can be resumed. It counts the documents in the output corpora and checks what the last written document was. It then updates the state to mark the module as partially executed, so that it continues from this document when you next try to run it.

The last written document is always thrown away, since we don't know whether it was fully written. To avoid partial, broken output, we assume the last document was not completed and resume execution on that one.

Note that this will only work for modules that output something (which may be an invalid doc) to every output for every input doc. Modules that only output to some outputs for each input cannot be recovered so easily.

Usage:

```
pimlico.sh [...] recover module [-h] [--dry] [--last-docs LAST_DOCS]
```

Positional arguments

Arg	Description
`module`	The name (or number) of the module to recover

Options

Option	Description
`--dry`	Dry run: just say what we'd do
`--last-docs`	Number of last docs to look at in each corpus when synchronizing

1.4.5 fixlength

Command-line tool subcommand

Under some circumstances (e.g. some unpredictable combinations of failures and restarts), an output corpus can end up with an incorrect length in its metadata. This command counts up the documents in the corpus and corrects the stored length if it's wrong.

Usage:

```
pimlico.sh [...] fixlength module [outputs [outputs ...]] [-h] [--dry]
```

Positional arguments

Arg	Description
`module`	The name (or number) of the module to recover
`[outputs [outputs ...]]`	Names of module outputs to check. By default, checks all

Options

Option	Description
`--dry`	Dry run: check the lengths, but don't write anything

1.4.6 browse

Command-line tool subcommand

View the data output by a module.

Usage:

```
pimlico.sh [...] browse module_name [output_name] [-h] [--skip-invalid] [--formatter␣
↪FORMATTER]
```

Positional arguments

Arg	Description
`module_name`	The name (or number) of the module whose output to look at. Use 'module:stage' for multi-stage modules
`[output_name]`	The name of the output from the module to browse. If blank, load the default output

Options

Option	Description
`--skip-invalid`	Skip over invalid documents, instead of showing the error that caused them to be invalid
`--formatter -f`	When browsing iterable corpora, fully qualified class name of a subclass of DocumentBrowserFormatter to use to determine what to output for each document. You may also choose from the named standard formatters for the datatype in question. Use '-f help' to see a list of available formatters

1.4.7 shell

Command-line tool subcommand

Open a shell to give access to the data output by a module.

Usage:

```
pimlico.sh [...] shell module_name [output_name] [-h]
```

Positional arguments

Arg	Description
`module_name`	The name (or number) of the module whose output to look at
`[output_name]`	The name of the output from the module to browse. If blank, load the default output

1.4.8 python

Command-line tool subcommand

Load the pipeline config and enter a Python interpreter with access to it in the environment.

Usage:

```
pimlico.sh [...] python [script] [-h] [-i]
```

Positional arguments

Arg	Description
`[script]`	Script file to execute. Omit to enter interpreter

Options

Option	Description
`-i`	Enter interactive shell after running script

1.4.9 reset

Command-line tool subcommand

Delete any output from the given module and restore it to unexecuted state.

Usage:

```
pimlico.sh [...] reset [modules [modules ...]] [-h] [-n] [-f]
```

Positional arguments

Arg	Description
`[modules [modules ...]]`	The names (or numbers) of the modules to reset, or 'all' to reset the whole pipeline

Options

Option	Description
`-n`, `--no-deps`	Only reset the state of this module, even if it has dependent modules in an executed state, which could be invalidated by resetting and re-running this one
`-f`, `--force-dep`	Reset the state of this module and any dependent modules in an executed state, which could be invalidated by resetting and re-running this one. Do not ask for confirmation to do this

1.4.10 clean

Command-line tool subcommand

Cleans up module output directories that have got left behind.

Often, when developing a pipeline incrementally, you try out some modules, but then remove them, or rename them to something else. The directory in the Pimlico output store that was created to contain their metadata, status and output data is then left behind and no longer associated with any module.

Run this command to check all storage locations for such directories. If it finds any, it prompts you to confirm before deleting them. (If there are things in the list that don't look like they were left behind by the sort of things mentioned above, don't delete them! I don't want you to lose your precious output data if I've made a mistake in this command.)

Note that the operation of this command is specific to the loaded pipeline variant. If you have multiple variants, make sure to select the one you want to clean with the general *–variant* option.

Usage:

```
pimlico.sh [...] clean [-h]
```

1.4.11 stores

Command-line tool subcommand

List Pimlico stores in use and the corresponding storage locations.

Usage:

```
pimlico.sh [...] stores [-h]
```

1.4.12 movestores

Command-line tool subcommand

Move a particular module's output from one storage location to another.

Usage:

```
pimlico.sh [...] movestores dest [modules [modules ...]] [-h]
```

Positional arguments

Arg	Description
`dest`	Name of destination store
`[modules [modules ...]]`	The names (or numbers) of the module whose output to move

1.4.13 unlock

Command-line tool subcommand

Forcibly remove an execution lock from a module. If a lock has ended up getting left on when execution exited prematurely, use this to remove it.

When a module starts running, it is locked to avoid making a mess of your output data by running the same module from another terminal, or some other silly mistake (I know, for some of us this sort of behaviour is frustratingly common).

Usually shouldn't be necessary, even if there's an error during execution, since the module should be unlocked when Pimlico exits, but occasionally (e.g. if you have to forcibly kill Pimlico during execution) the lock gets left on.

Usage:

```
pimlico.sh [...] unlock module_name [-h]
```

Positional arguments

Arg	Description
`module_name`	The name (or number) of the module to unlock

1.4.14 dump

Command-line tool subcommand

Dump the entire available output data from a given pipeline module to a tarball, so that it can easily be loaded into the same pipeline on another system. This is primarily to support spreading the execution of a pipeline between multiple machines, so that the output from a module can easily be transferred and loaded into a pipeline.

Dump to a tarball using this command, transfer the file between machines and then run the *load command* to import it there.

See also:

Running on multiple computers: for a more detailed guide to transferring data across servers.

Usage:

```
pimlico.sh [...] dump [modules [modules ...]] [-h] [--output OUTPUT] [--inputs]
```

Positional arguments

Arg	Description
`[modules [modules ...]]`	Names or numbers of modules whose data to dump. If multiple are given, a separate file will be dumped for each

Options

Option	Description
`--output, -o`	Path to directory to output to. Defaults to the current user's home directory
`--inputs, -i`	Dump data for the modules corresponding to the inputs of the named modules, instead of those modules themselves. Useful for when you're preparing to run a module on a different machine, for getting all the necessary input data for a module

1.4.15 load

Command-line tool subcommand

Load the output data for a given pipeline module from a tarball previously created by the *dump* command (typically on another machine). This is primarily to support spreading the execution of a pipeline between multiple machines, so that the output from a module can easily be transferred and loaded into a pipeline.

Dump to a tarball using the *dump command*, transfer the file between machines and then run this command to import it there.

See also:

Running on multiple computers: for a more detailed guide to transferring data across servers.

Usage:

```
pimlico.sh [...] load [paths [paths ...]] [-h] [--force-overwrite]
```

Positional arguments

Arg	Description
`[paths [paths ...]]`	Paths to dump files (tarballs) to load into the pipeline

Options

Option	Description
`--force-overwrite, -f`	If data already exists for a module being imported, overwrite without asking. By default, the user will be prompted to check whether they want to overwrite

1.4.16 deps

Command-line tool subcommand

Output information about module dependencies.

Usage:

```
pimlico.sh [...] deps [modules [modules ...]] [-h]
```

Positional arguments

Arg	Description
`[modules [modules ... ]]`	Check dependencies for named modules and install any that are automatically installable. Use 'all' to install dependencies for all modules

1.4.17 install

Command-line tool subcommand

Install missing dependencies.

Usage:

```
pimlico.sh [...] install [modules [modules ...]] [-h] [--trust-downloaded]
```

Positional arguments

Arg	Description
`[modules [modules ... ]]`	Check dependencies for named modules and install any that are automatically installable. Use 'all' to install dependencies for all modules

Options

Option	Description
`--trust-downloaded, -t`	If an archive file to be downloaded is found to be in the lib dir already, trust that it is the file we're after. By default, we only reuse archives we've just downloaded, so we know they came from the right URL, avoiding accidental name clashes

1.4.18 inputs

Command-line tool subcommand

Show the locations of the inputs of a given module. If the input datasets are available, their actual location is shown. Otherwise, all directories in which the data is being checked for are shown.

Usage:

```
pimlico.sh [...] inputs module_name [-h]
```

Positional arguments

Arg	Description
`module_name`	The name (or number) of the module to display input locations for

1.4.19 output

Command-line tool subcommand

Show the location where the given module's output data will be (or has been) stored.

Usage:

```
pimlico.sh [...] output module_name [-h]
```

Positional arguments

Arg	Description
`module_name`	The name (or number) of the module to display input locations for

1.4.20 newmodule

Command-line tool subcommand

Interactive tool to create a new module type, generating a skeleton for the module's code. Currently only works for certain module types. May be extended in future to help with creating a broader range of sorts of modules.

Usage:

```
pimlico.sh [...] newmodule [-h]
```

1.4.21 visualize

Command-line tool subcommand

(Not yet fully implemented!) Visualize the pipeline, with status information for modules.

Usage:

```
pimlico.sh [...] visualize [-h] [--all]
```

Options

Option	Description
`--all`, `-a`	Show all modules defined in the pipeline, not just those that can be executed

1.4.22 email

Command-line tool subcommand

Test email settings and try sending an email using them.

Usage:

```
pimlico.sh [...] email [-h]
```

1.4.23 jupyter

Command-line tool subcommand

Creates and runs a Jupyter notebook for the loaded pipeline. The pipeline is made easily available within the notebook, providing a way to load the modules and get their outputs.

This is a useful way to explore the data or analyses coming out of your modules. Once a module has been run, you can load it from a notebook and manipulate, explore, visualize, etc to results.

A new directory is automatically created in your project root to contain the pipeline's notebooks. (You can override the location of this using `--notebook-dir`). An example notebook is created there, to show you how to load the pipeline.

From within a notebook, load a pipeline like so:

```
from pimlico import get_jupyter_pipeline
pipeline = get_jupyter_pipeline()
```

Now you can access the modules of the pipeline through this pipeline object:

```
mod = pipeline["my_module"]
```

And get data from its outputs (provided the module's been run):

```
print(mod.status)
output = mod.get_output("output_name").
```

Usage:

```
pimlico.sh [...] jupyter [-h] [--notebook-dir NOTEBOOK_DIR]
```

Options

Option	Description
`--notebook-dir`	Use a custom directory as the notebook directory. By default, a directory will be created according to: <pimlico_root>/../notebooks/<pipeline_name>/

1.4.24 tar2pimarc

Command-line tool subcommand

Convert grouped corpora from the old tar-based storage format to Pimarc archives.

Usage:

```
pimlico.sh [...] tar2pimarc [outputs [outputs ...]] [-h] [--run]
```

Positional arguments

Arg	Description
`[outputs [outputs ...]]`	Specification of module outputs to convert. Specific datasets can be given as 'module_name.output_name'. All grouped corpus outputs of a module can be converted by just giving 'module_name'. Or, if nothing's given, all outputs of all modules are converted

Options

Option	Description
`--run`	Run conversion. Without this option, just checks what format the corpora use

1.4.25 licenses

Command-line tool subcommand

Output a list of the licenses for all software depended on.

Usage:

```
pimlico.sh [...] licenses [modules [modules ...]] [-h]
```

Positional arguments

Arg	Description
`[modules [modules ...]]`	Check dependencies of modules and their datatypes. Use 'all' to list licenses for dependencies for all modules

1.5 API Documentation

API documentation for the main Pimlico codebase, excluding the *built-in Pimlico module types*.

1.5.1 pimlico

Subpackages

cli

Subpackages

browser

Subpackages

tools

Submodules

corpus

Browser tool for iterable corpora.

browse_data(*reader*, *formatter*, *skip_invalid=False*)

class CorpusState(*corpus*)

Bases: `object`

Keep track of which document we're on.

next_document()

skip(*n*)

class InputDialog(*text*, *input_edit*)
Bases: urwid.widget.WidgetWrap

A dialog that appears with an input

signals = ['close', 'cancel']

keypress(*size*, *k*)

class MessageDialog(*text*, *default=None*)
Bases: urwid.widget.WidgetWrap

A dialog that appears with a message

class InputPopupLauncher(*original_widget*, *text*, *input_edit*, *callback=None*)
Bases: urwid.wimp.PopUpLauncher

create_pop_up()
Subclass must override this method and return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

get_pop_up_parameters()
Subclass must override this method and have it return a dict, eg:

{'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

This method is called each time this widget is rendered.

skip_popup_launcher(*original_widget*, *text*, *default=None*, *callback=None*)

save_popup_launcher(*original_widget*, *text*, *default=None*, *callback=None*)

class MessagePopupLauncher(*original_widget*, *text*)
Bases: urwid.wimp.PopUpLauncher

create_pop_up()
Subclass must override this method and return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

get_pop_up_parameters()
Subclass must override this method and have it return a dict, eg:

{'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

This method is called each time this widget is rendered.

files

browse_files(*reader*)
Browser tool for NamedFileCollections.

is_binary_string(*bytes*)

is_binary_file(*path*)
Try reading a bit of a file to work out whether it's a binary file or text

formatter

The command-line iterable corpus browser displays one document at a time. It can display the raw data from the corpus files, which sometimes is sufficiently human-readable to not need any special formatting. It can also parse the

data using its datatype and output text either from the datatype's standard unicode representation or, if the document datatype provides it, a special browser formatting of the data.

When viewing output data, particularly during debugging of modules, it can be useful to provide special formatting routines to the browser, rather than using or overriding the datatype's standard formatting methods. For example, you might want to pull out specific attributes for each document to get an overview of what's coming out.

The browser command accepts a command-line option that specifies a Python class to format the data. This class should be a subclass of :class:~pimlico.cli.browser.formatter.DocumentBrowserFormatter that accepts a datatype compatible with the datatype being browsed and provides a method to format each document. You can write these in your custom code and refer to them by their fully qualified class name.

class DocumentBrowserFormatter(*corpus_datatype*)

Bases: `object`

Base class for formatters used to post-process documents for display in the iterable corpus browser.

DATATYPE = DataPointType()

format_document(*doc*)

Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

filter_document(*doc*)

Each doc is passed through this function directly after being read from the corpus. If None is returned, the doc is skipped. Otherwise, the result is used instead of the doc data. The default implementation does nothing.

class DefaultFormatter(*corpus_datatype*)

Bases: *pimlico.cli.browser.tools.formatter.DocumentBrowserFormatter*

Generic implementation of a browser formatter that's used if no other formatter is given.

DATATYPE = DataPointType()

format_document(*doc*)

Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class InvalidDocumentFormatter(*corpus_datatype*)

Bases: *pimlico.cli.browser.tools.formatter.DocumentBrowserFormatter*

Formatter that skips over all docs other than invalid results. Uses standard formatting for InvalidDocument information.

format_document(*doc*)

Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

filter_document(*doc*)

Each doc is passed through this function directly after being read from the corpus. If None is returned, the doc is skipped. Otherwise, the result is used instead of the doc data. The default implementation does nothing.

typecheck_formatter(*formatted_doc_type*, *formatter_cls*)

Check that a document type is compatible with a particular formatter.

load_formatter(*datatype*, *formatter_name=None*)
Load a formatter specified by its fully qualified Python class name. If None, loads the default formatter. You may also specify a formatter by name, choosing from one of the standard ones that the formatted datatype gives.

Parameters

- **datatype** – datatype instance representing the datatype that will be formatted
- **formatter_name** – class name, or class

Returns instantiated formatter

Module contents

Submodules

tool

Tool for browsing datasets, reading from the data output by pipeline modules.

browse_cmd(*pipeline*, *opts*)
Command for main Pimlico CLI

Module contents

data_editor

Submodules

pimlico.cli.data_editor.run module

run_editor(*dataset_root*, *datatype_name*)

Module contents

A separate command-line tool used to edit datasets on disk.

This is a new addition and not very mature yet. Its main purpose is to make it easy to create test datasets manually without having to write special pipelines.

debug

Submodules

stepper

class Stepper
Bases: `object`

Type that stores the state of the stepping process. This allows information and parameters to be passed around through the process and updated as we go. For example, if particular type of output is disabled by the user, a parameter can be updated here so we know not to output it later.

enable_step_for_pipeline(*pipeline*)
Prepares a pipeline to run in step mode, modifying modules and wrapping methods to supply the extra functionality.

This approach means that we don't have to consume extra computation time checking whether step mode is enabled during normal runs.

Parameters **pipeline** – instance of PipelineConfig

instantiate_output_reader_decorator(*instantiate_output_reader*, *module_name*, *output_names*, *stepper*)

wrap_grouped_corpus(*dtype*, *module_name*, *output_name*, *stepper*)

archive_iter_decorator(*archive_iter*, *module_name*, *output_name*, *stepper*)

get_input_decorator(*get_input*, *module_name*, *stepper*)
Decorator to wrap a module info's get_input() method so when know where inputs are being used.

option_message(*message_lines*, *stepper*, *options=None*, *stack_trace_option=True*, *category=None*)

Module contents

Extra-verbose debugging facility

Tools for very slowly and verbosely stepping through the processing that a given module does to debug it.

Enabled using the *–step* switch to the run command.

fmt_frame_info(*info*)

output_stack_trace(*frame=None*)

shell

Submodules

base

class ShellCommand
Bases: `object`

Base class used to provide commands for exploring a particular datatype. A basic set of commands is provided for all datatypes, but specific datatype classes may provide their own, by overriding the *shell_commands* attribute.

commands = []

help_text = None

execute(*shell*, **args*, ***kwargs*)
Execute the command. Get the dataset reader as shell.data.

Parameters

- **shell** – DataShell instance. Reader available as shell.data

- **args** – Args given by the user
- **kwargs** – Named args given by the user as key=val

class DataShell(*data*, *commands*, **args*, ***kwargs*)
Bases: cmd.Cmd

Terminal shell for querying datatypes.

prompt = '>>> '

get_names()

do_EOF(*line*)
Exits the shell

preloop()
Hook method executed once when the cmdloop() method is called.

postloop()
Hook method executed once when the cmdloop() method is about to return.

emptyline()
Don't repeat the last command (default): ignore empty lines

default(*line*)
We use this to handle commands that can't be handled using the *do_* pattern. Also handles the default fallback, which is to execute Python.

cmdloop(*intro=None*)
Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

exception ShellError
Bases: Exception

commands

Basic set of shell commands that are always available.

class MetadataCmd
Bases: *pimlico.cli.shell.base.ShellCommand*

commands = ['metadata']

help_text = "Display the loaded dataset's metadata"

execute(*shell*, **args*, ***kwargs*)
Execute the command. Get the dataset reader as shell.data.

Parameters

- **shell** – DataShell instance. Reader available as shell.data
- **args** – Args given by the user
- **kwargs** – Named args given by the user as key=val

class PythonCmd
Bases: *pimlico.cli.shell.base.ShellCommand*

commands = ['python', 'py']

help_text = "Run a Python interpreter using the current environment, including import

execute (*shell*, **args*, ***kwargs*)
Execute the command. Get the dataset reader as shell.data.

Parameters

- **shell** – DataShell instance. Reader available as shell.data
- **args** – Args given by the user
- **kwargs** – Named args given by the user as key=val

runner

class ShellCLICmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'shell'

command_help = 'Open a shell to give access to the data output by a module'

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

launch_shell (*data*)
Starts a shell to view and query the given datatype instance.

Module contents

Submodules

check

class InstallCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Install missing dependencies.

command_name = 'install'

command_help = 'Install missing module library dependencies'

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

class DepsCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Output information about module dependencies.

command_name = 'deps'

command_help = "List information about software dependencies: whether they're availab

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

class LicensesCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Output a list of the licenses for all software depended on.

command_name = 'licenses'

command_help = 'List information about licsenses of software dependencies'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

clean

class CleanCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Cleans up module output directories that have got left behind.

Often, when developing a pipeline incrementally, you try out some modules, but then remove them, or rename them to something else. The directory in the Pimlico output store that was created to contain their metadata, status and output data is then left behind and no longer associated with any module.

Run this command to check all storage locations for such directories. If it finds any, it prompts you to confirm before deleting them. (If there are things in the list that don't look like they were left behind by the sort of things mentioned above, don't delete them! I don't want you to lose your precious output data if I've made a mistake in this command.)

Note that the operation of this command is specific to the loaded pipeline variant. If you have multiple variants, make sure to select the one you want to clean with the general *–variant* option.

command_name = 'clean'

command_help = 'Remove all module directories that do not correspond to a module in th

command_desc = 'Remove all module output directories that do not correspond to a modul

run_command(*pipeline*, *opts*)

fixlength

class FixLengthCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Under some circumstances (e.g. some unpredictable combinations of failures and restarts), an output corpus can end up with an incorrect length in its metadata. This command counts up the documents in the corpus and corrects the stored length if it's wrong.

command_name = 'fixlength'

command_help = "Check the length of written outputs and fix it if it's wrong"

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

count_pimarcs(*output*)

jupyter

A command to start a Jupyter notebook for a given pipeline, providing access to its modules and their outputs.

class JupyterCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Creates and runs a Jupyter notebook for the loaded pipeline. The pipeline is made easily available within the notebook, providing a way to load the modules and get their outputs.

This is a useful way to explore the data or analyses coming out of your modules. Once a module has been run, you can load it from a notebook and manipulate, explore, visualize, etc to results.

A new directory is automatically created in your project root to contain the pipeline's notebooks. (You can override the location of this using `--notebook-dir`). An example notebook is created there, to show you how to load the pipeline.

From within a notebook, load a pipeline like so:

```
from pimlico import get_jupyter_pipeline
pipeline = get_jupyter_pipeline()
```

Now you can access the modules of the pipeline through this pipeline object:

```
mod = pipeline["my_module"]
```

And get data from its outputs (provided the module's been run):

```
print(mod.status)
output = mod.get_output("output_name")
```

command_name = 'jupyter'

command_help = 'Create and start a new Jupyter notebook for the pipeline'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

make_notebook(*code_text*)

loaddump

class DumpCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Dump the entire available output data from a given pipeline module to a tarball, so that it can easily be loaded into the same pipeline on another system. This is primarily to support spreading the execution of a pipeline between multiple machines, so that the output from a module can easily be transferred and loaded into a pipeline.

Dump to a tarball using this command, transfer the file between machines and then run the *load command* to import it there.

See also:

Running on multiple computers: for a more detailed guide to transferring data across servers

command_name = 'dump'

command_help = 'Dump the entire available output data from a given pipeline module to a

command_desc = 'Dump the entire available output data from a given pipeline module to a

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class LoadCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Load the output data for a given pipeline module from a tarball previously created by the *dump* command (typically on another machine). This is primarily to support spreading the execution of a pipeline between multiple machines, so that the output from a module can easily be transferred and loaded into a pipeline.

Dump to a tarball using the *dump command*, transfer the file between machines and then run this command to import it there.

See also:

Running on multiple computers: for a more detailed guide to transferring data across servers

command_name = 'load'

command_help = "Load a module's output data from a tarball previously created by the d

command_desc = "Load a module's output data from a tarball previously created by the d

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

locations

class InputsCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'inputs'

command_help = 'Show the locations of the inputs of a given module. If the input datas

command_desc = 'Show the (expected) locations of the inputs of a given module'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class OutputCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'output'

command_help = "Show the location where the given module's output data will be (or has

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class ListStoresCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'stores'

command_help = 'List Pimlico stores in use and the corresponding storage locations'

command_desc = 'List named Pimlico stores'

run_command(*pipeline*, *opts*)

class MoveStoresCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'movestores'

command_help = "Move a particular module's output from one storage location to another

command_desc = 'Move data between stores'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

main

Main command-line script for running Pimlico, typically called from *pimlico.sh*.

Provides access to many subcommands, acting as the primary interface to Pimlico's functionality.

class VariantsCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

List the available variants of a pipeline config

See *Pipeline variants* for more details.

command_name = 'variants'

command_help = 'List the available variants of a pipeline config'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class UnlockCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Forcibly remove an execution lock from a module. If a lock has ended up getting left on when execution exited prematurely, use this to remove it.

When a module starts running, it is locked to avoid making a mess of your output data by running the same module from another terminal, or some other silly mistake (I know, for some of us this sort of behaviour is frustratingly common).

Usually shouldn't be necessary, even if there's an error during execution, since the module should be unlocked when Pimlico exits, but occasionally (e.g. if you have to forcibly kill Pimlico during execution) the lock gets left on.

command_name = 'unlock'

command_help = "Forcibly remove an execution lock from a module. If a lock has ended u

command_desc = 'Forcibly remove an execution lock from a module'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class BrowseCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'browse'

command_help = 'View the data output by a module'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class VisualizeCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'visualize'

command_help = '(Not yet fully implemented!) Visualize the pipeline, with status infor

command_desc = 'Comming soon...visualize the pipeline in a pretty way'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

newmodule

class NewModuleCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'newmodule'

command_help = "Interactive tool to create a new module type, generating a skeleton fo

command_desc = 'Create a new module type'

run_command(*pipeline*, *opts*)

ask(*prompt*, *strip_space=True*)

pimarc

class Tar2PimarcCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Convert grouped corpora from the old tar-based storage format to Pimarc archives.

command_name = 'tar2pimarc'

command_help = 'Convert grouped corpora from the old tar-based storage format to pimar

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

tar_to_pimarc(*in_tar_paths*)

pyshell

class PimlicoPythonShellContext

Bases: `object`

A class used as a static global data structure to provide access to the loaded pipeline when running the Pimlico Python shell command.

This should never be used in any other context to pass around loaded pipelines or other global data. We don't do that sort of thing.

class PythonShellCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'python'

command_help = 'Load the pipeline config and enter a Python interpreter with access to

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

get_pipeline()
This function may be used in scripts that are expected to be run exclusively from the Pimlico Python shell command (`python`) to get hold of the pipeline that was specified on the command line and loaded when the shell was started.

exception ShellContextError
Bases: `Exception`

recover

class RecoverCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

When a document map module gets killed forcibly, sometimes it doesn't have time to save its execution state, meaning that it can't pick up from where it left off.

Todo: This has not been updated for the Pimarc internal storage format, so still assumes that tar files are used. It will be updated in future, if there is a need for it.

This command tries to fix the state so that execution can be resumed. It counts the documents in the output corpora and checks what the last written document was. It then updates the state to mark the module as partially executed, so that it continues from this document when you next try to run it.

The last written document is always thrown away, since we don't know whether it was fully written. To avoid partial, broken output, we assume the last document was not completed and resume execution on that one.

Note that this will only work for modules that output something (which may be an invalid doc) to every output for every input doc. Modules that only output to some outputs for each input cannot be recovered so easily.

command_name = 'recover'

command_help = "Examine and fix a partially executed map module's output state after f

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

count_docs(*corpus*, *last_buffer_size=10*)

truncate_tar_after(*path*, *last_filename*, *gzipped=False*)
Read through the given tar file to find the specified filename. Truncate the archive after the end of that file's contents.

Creates a backup of the tar archive first, since this is a risky operation.

Returns False if the filename wasn't found

reset

class ResetCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'reset'

command_help = 'Delete any output from the given module and restore it to unexecuted s

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

run

class RunCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Main command for executing Pimlico modules from the command line *run* command.

command_name = 'run'

command_help = 'Execute an individual pipeline module, or a sequence'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

status

class StatusCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'status'

command_help = 'Output a module execution schedule for the pipeline and execution statu

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

print_section_tree(*tree*, *mod_name_bullets*, *pipeline*, *depth=0*, *expand='all'*, *aliases=None*)

print_module_status(*module_name*, *bullet*, *pipeline*, *aliases=None*)

module_status_color(*module*)

status_colored(*module*, *text=None*)

Colour the text according to the status of the given module. If text is not given, the module's name is returned.

mix_bg_colors(*text*, *colors*)

Format a string with mixed colors, by alternating by character.

module_status(*module*)

Detailed module status, shown when a specific module's status is requested.

subcommands

class PimlicoCLISubcommand

Bases: `object`

Base class for defining subcommands to the main command line tool.

This allows us to split up subcommands, together with all their arguments/options and their functionality, since there are quite a lot of them.

Documentation of subcommands should be supplied in the following ways:

- Include help texts for positional args and options in the add_arguments() method. They will all be included in the doc page for the command.
- Write a very short description of what the command is for (a few words) in command_desc. This will be used in the summary table / TOC in the docs.
- Write a short description of what the command does in `command_help`. This will be available in command-line help and used as a fallback if you don't do the next point.
- Write a good guide to using the command (or at least say what it does) in the class' docstring (i.e. overriding this). This will form the bulk of the command's doc page.

command_name = None

command_help = None

command_desc = None

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

testemail

class EmailCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'email'

command_help = 'Test email settings and try sending an email using them'

run_command(*pipeline*, *opts*)

util

module_number_to_name(*pipeline*, *name*)

module_numbers_to_names(*pipeline*, *names*)

Convert module numbers to names, also handling ranges of numbers (and names) specified with "...". Any "..." will be filled in by the sequence of intervening modules.

Also, if an unexpanded module name is specified for a module that's been expanded into multiple corresponding to alternative parameters, all of the expanded module names are inserted in place of the unexpanded name.

format_execution_error(*error*)

Produce a string with lots of error output to help debug a module execution error.

Parameters **error** – the exception raised (ModuleExecutionError or ModuleInfoLoadError)

Returns formatted output

print_execution_error(*error*)

Module contents

core

Subpackages

dependencies

Submodules

base

Base classes for defining software dependencies for module types and routines for fetching them.

class SoftwareDependency(*name*, *homepage_url=None*, *dependencies=None*, *license=None*)
Bases: `object`

Base class for all Pimlico module software dependencies.

Every dependency has a name and list of sub-dependencies.

A URL may be provided by the kwarg `homepage_url`. This will be used in documentation to link to the software's homepage.

A license may also be specified, for inclusion in the documentation. It should be an instance of `SoftwareLicense`. See the literals in `pimlico.core.dependences.licenses` for many commonly used licenses.

available(*local_config*)
Return True if the dependency is satisfied, meaning that the software/library is installed and ready to use.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

installation_instructions()
Where a dependency can't be installed programmatically, we typically want to be able to output instructions for the user to tell them how to go about doing it themselves. Any subclass that doesn't provide an automatic installation routine should override this to provide instructions.

You may also provide this even if the class does provide automatic installation. For example, you might want to provide instructions for other ways to install the software, like a system-wide install. This instructions will be shown together with missing dependency information.

installation_notes()
If this returns a non-empty string, the message will be output together with the information that the dependency is not available, before the user is given the option of installing it automatically (or told that it can't be). This is useful where information about a dependency should always be displayed, not just in cases where automatic installation isn't possible.

For example, you might need to include warnings about potential installation difficulties, license information, sources of additional information about the software, and so on.

dependencies()
Returns a list of instances of *SoftwareDependency* subclasses representing this library's own depen-

dencies. If the library is already available, these will never be consulted, but if it is to be installed, we will check first that all of these are available (and try to install them if not).

install(*local_config*, *trust_downloaded_archives=False*)
Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

all_dependencies()
Recursively fetch all dependencies of this dependency (not including itself).

get_installed_version(*local_config*)
If available() returns True, this method should return a SoftwareVersion object (or subclass) representing the software's version.

The base implementation returns an object representing an unknown version number.

If available() returns False, the behaviour is undefined and may raise an error.

class Any(*name*, *dependency_options*, **args*, ***kwargs*)
Bases: *pimlico.core.dependencies.base.SoftwareDependency*

A collection of dependency requirements of which at least one must be available. The first in the list that is installable is treated as the default and used for automatic installation.

available(*local_config*)
Return True if the dependency is satisfied, meaning that the software/library is installed and ready to use.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

get_installation_candidate()
Returns the first dependency of the multiple possibilities that is automatically installable, or None if none of them are.

get_available_option(*local_config*)
If one of the options is available, return that one. Otherwise return None.

dependencies()
Returns a list of instances of *SoftwareDependency* subclasses representing this library's own dependencies. If the library is already available, these will never be consulted, but if it is to be installed, we will check first that all of these are available (and try to install them if not).

install(*local_config*, *trust_downloaded_archives=False*)
Installs the dependency given by *get_installation_candidate()*, if any. Ideally, we should provide a way to select which of the options should be installed. However, until we've worked out the best way to do this, the default option is always installed. The user may install another option manually and that will be used.

installation_notes()
If this returns a non-empty string, the message will be output together with the information that the dependency is not available, before the user is given the option of installing it automatically (or told that it can't be). This is useful where information about a dependency should always be displayed, not just in cases where automatic installation isn't possible.

For example, you might need to include warnings about potential installation difficulties, license information, sources of additional information about the software, and so on.

class SystemCommandDependency(*name*, *test_command*, ***kwargs*)
Bases: *pimlico.core.dependencies.base.SoftwareDependency*

Dependency that tests whether a command is available on the command line. Generally requires system-wide installation.

installable()
Usually not automatically installable

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

exception InstallationError
Bases: Exception

check_and_install(*deps*, *local_config*, *trust_downloaded_archives=False*)
Check whether dependencies are available and try to install those that aren't. Returns a list of dependencies that can't be installed.

install(*dep*, *local_config*, *trust_downloaded_archives=False*)

install_dependencies(*pipeline*, *modules=None*, *trust_downloaded_archives=True*)
Install depedencies for pipeline modules

Parameters

- **pipeline** –
- **modules** – list of module names, or None to install for all

Returns

recursive_deps(*dep*)
Collect all recursive dependencies of this dependency. Does a depth-first search so that everything comes later in the list than things it depends on.

core

Basic Pimlico core dependencies

CORE_PIMLICO_DEPENDENCIES = [PythonPackageOnPip<virtualenv>, PythonPackageOnPip<colorama>,
Core dependencies required by the basic Pimlico installation, regardless of what pipeline is being processed. These will be checked when Pimlico is run, using the same dependency-checking mechanism that Pimlico modules use, and installed automatically if they're not found.

java

class JavaDependency(*name*, *classes=[]*, *jars=[]*, *class_dirs=[]*, ***kwargs*)
Bases: *pimlico.core.dependencies.base.SoftwareDependency*

Base class for Java library dependencies.

In addition to the usual functionality provided by dependencies, subclasses of this provide contributions to the Java classpath in the form of directories of jar files.

The instance has a set of representative Java classes that the checker will try to load to check whether the library is available and functional. It will also check that all jar files exist.

Jar paths and class directory paths are assumed to be relative to the Java lib dir (lib/java), unless they are absolute paths.

Subclasses should provide install() and override installable() if it's possible to install them automatically.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

jar_paths(*local_config*)
Absolute paths to the jars

all_jars(*local_config*)
Get all jars, including from dependencies

get_classpath_components()

class JavaJarsDependency(*name*, *jar_urls*, ***kwargs*)
Bases: *pimlico.core.dependencies.java.JavaDependency*

Simple way to define a Java dependency where the library is packaged up in a jar, or a series of jars. The jars should be given as a list of (name, url) pairs, where name is the filename the jar should have and url is a url from which it can be downloaded.

URLs may also be given in the form "url->member", where url is a URL to a tar.gz or zip archive and member is a member to extract from the archive. If the type of the file isn't clear from the URL (i.e. if it doesn't have ".zip" or ".tar.gz" in it), specify the intended extension in the form "[ext]url->member", where ext is "tar.gz" or "zip".

If multiple jars come from the same URL (i.e. the same archive), it will only be downloaded once.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install (*local_config*, *trust_downloaded_archives=False*)
Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

class PimlicoJavaLibrary (*name*, *classes=[]*, *additional_jars=[]*)
Bases: *pimlico.core.dependencies.java.JavaDependency*

Special type of Java dependency for the Java libraries provided with Pimlico. These are packages up in jars and stored in the build dir.

check_java_dependency (*class_name*, *classpath=None*)
Utility to check that a java class is able to be loaded.

check_java ()
Check that the JVM executable can be found. Raises a DependencyError if it can't be found or can't be run.

get_classpath (*deps*, *as_list=False*)
Given a list of JavaDependency subclass instances, returns all the components of the classpath that will make sure that the dependencies are available.

If as_list=True, returned as a list. Get the full classpath by ":".join(x) on the list. If as_list=False, returns classpath string.

get_module_classpath (*module*)
Builds a classpath that includes all of the classpath elements specified by Java dependencies of the given module. These include the dependencies from get_software_dependencies() and also any dependencies of the datatype.

Used to ensure that Java modules that depend on particular jars or classes get all of those files included on their classpath when Java is run.

class Py4JSoftwareDependency
Bases: *pimlico.core.dependencies.java.JavaDependency*

Java component of Py4J. Use this one as the main dependency, as it depends on the Python component and will install that first if necessary.

dependencies ()
Returns a list of instances of `SoftwareDependency` subclasses representing this library's own dependencies. If the library is already available, these will never be consulted, but if it is to be installed, we will check first that all of these are available (and try to install them if not).

jars

installable ()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install (*local_config*, *trust_downloaded_archives=False*)
Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

licenses

Software licenses, for referring to in software dependency documentation.

Literals here are used to refer to the licenses that software uses.

See https://choosealicense.com/licenses/ for more details and comparison.

class SoftwareLicense(*name*, *description=None*, *url=None*)
Bases: `object`

python

Tools for Python library dependencies.

Provides superclasses for Python library dependencies and a selection of commonly used dependency instances.

class PythonPackageDependency(*package*, *name*, ***kwargs*)
Bases: *pimlico.core.dependencies.base.SoftwareDependency*

Base class for Python dependencies. Provides import checks, but no installation routines. Subclasses should either provide install() or installation_instructions().

The import checks do not (as of 0.6rc) actually import the package, as this may have side-effects that are difficult to account for, causing odd things to happen when you check multiple times, or try to import later. Instead, it just checks whether the package finder is about to locate the package. This doesn't guarantee that the import will succeed.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

import_package()
Try importing package_name. By default, just uses *__import__*. Allows subclasses to allow for special import behaviour.

Should raise an *ImportError* if import fails.

get_installed_version(*local_config*)
Tries to import a __version__ variable from the package, which is a standard way to define the package version.

class PythonPackageSystemwideInstall(*package_name*, *name*, *pip_package=None*, *apt_package=None*, *yum_package=None*, ***kwargs*)
Bases: *pimlico.core.dependencies.python.PythonPackageDependency*

Dependency on a Python package that needs to be installed system-wide.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

installation_instructions()
Where a dependency can't be installed programmatically, we typically want to be able to output instructions for the user to tell them how to go about doing it themselves. Any subclass that doesn't provide an automatic installation routine should override this to provide instructions.

You may also provide this even if the class does provide automatic installation. For example, you might want to provide instructions for other ways to install the software, like a system-wide install. This instructions will be shown together with missing dependency information.

class PythonPackageOnPip(*package*, *name=None*, *pip_package=None*, *upgrade_only_if_needed=False*, *min_version=None*, *editable=False*, ***kwargs*)
Bases: *pimlico.core.dependencies.python.PythonPackageDependency*

Python package that can be installed via pip. Will be installed in the virtualenv if not available.

Allows specification of a minimum version. If an earlier version is installed, it will be upgraded.

Name is the readable software name. Package is a the package that is imported in Python.

Parameters **editable** (*boolean*) – Pass the –editable option to pip when installing. Use with e.g. Git urls as packages.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install(*local_config*, *trust_downloaded_archives=False*)
Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

get_installed_version(*local_config*)
Tries to import a __version__ variable from the package, which is a standard way to define the package version.

safe_import_bs4()
BS can go very slowly if it tries to use chardet to detect input encoding Remove chardet and cchardet from the Python modules, so that import fails and it doesn't try to use them This prevents it getting stuck on reading long input files

class BeautifulSoupDependency
Bases: *pimlico.core.dependencies.python.PythonPackageOnPip*

Test import with special BS import behaviour.

import_package()
Try importing package_name. By default, just uses *__import__*. Allows subclasses to allow for special import behaviour.

Should raise an *ImportError* if import fails.

class NLTKResource(*name*, *homepage_url=None*, *dependencies=None*, *license=None*)
Bases: *pimlico.core.dependencies.base.SoftwareDependency*

Check for and install NLTK resources, using NLTK's own downloader.

problems(*local_config*)
Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

installable()
Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install(*local_config*, *trust_downloaded_archives=False*)
Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

dependencies()
Returns a list of instances of `SoftwareDependency` subclasses representing this library's own dependencies. If the library is already available, these will never be consulted, but if it is to be installed, we will check first that all of these are available (and try to install them if not).

versions

class SoftwareVersion(*string_id*)
Bases: `object`

Base class for representing version numbers / IDs of software. Different software may use different conventions to represent its versions, so it may be necessary to subclass this class to provide the appropriate parsing and comparison of versions.

compare_dotted_versions(*version0*, *version1*)
Comparison function for reasonably standard version numbers, with subversions to any level of nesting specified by dots.

Module contents

external

Submodules

java

call_java(*class_name*, *args=[]*, *classpath=None*)

java_call_command(*class_name*, *classpath=None*)
List of components for a subprocess call to Java, used by call_java

start_java_process(*class_name*, *args=[]*, *java_args=[]*, *wait=0.1*, *classpath=None*)

class Py4JInterface(*gateway_class*, *port=None*, *python_port=None*, *gateway_args=[]*, *pipeline=None*, *print_stdout=True*, *print_stderr=True*, *env={}*, *system_properties={}*, *java_opts=[]*, *timeout=10.0*, *prefix_classpath=None*)

Bases: `object`

If pipeline is given, configuration is looked for there. If found, this overrides config given in other kwargs.

If print_stdout=True (default), stdout from processes will be printed out to the console in addition to any other processing that's done to it. Same with stderr. By default, both are output to the console.

env adds extra variables to the environment for running the Java process.

system_properties adds Java system property settings to the Java command.

start(*timeout=None*, *port_output_prefix=None*)

Start a Py4J gateway server in the background on the given port, which will then be used for communicating with the Java app.

If a port has been given, it is assumed that the gateway accepts a –port option. Likewise with python_port and a –python-port option.

If timeout is given, it overrides any timeout given in the constructor or specified in local config.

new_client()

stop()

clear_output_queues()

no_retry_gateway(***kwargs*)

A wrapper around the constructor of JavaGateway that produces a version of it that doesn't retry on errors. The default gateway keeps retying and outputting millions of errors if the server goes down, which makes responding to interrupts horrible (as the server might die before the Python process gets the interrupt).

TODO This isn't working: it just gets worse when I use my version!

gateway_client_to_running_server(*port*)

launch_gateway(*gateway_class='py4j.GatewayServer'*, *args=[]*, *javaopts=[]*, *redirect_stdout=None*, *redirect_stderr=None*, *daemonize_redirect=True*, *env={}*, *port_output_prefix=None*, *startup_timeout=10.0*, *prefix_classpath=None*)

Our own more flexble version of Py4J's launch_gateway.

get_redirect_func(*redirect*)

class OutputConsumer(*redirects*, *stream*, **args*, ***kwargs*)

Bases: `threading.Thread`

Thread that consumes output Modification of Py4J's OutputConsumer to allow multiple redirects.

remove_temporary_redirects()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

output_p4j_error_info(*command*, *returncode*, *stdout*, *stderr*)

make_py4j_errors_safe(*fn*)

Decorator for functions/methods that call Py4J. Py4J's exceptions include information that gets retrieved from

the Py4J server when they're displayed. This is a problem if the server is not longer running and raises another exception, making the whole situation very confusing.

If you wrap your function with this, Py4JJavaErrors will be replaced by our own exception type Py4JSafeJavaError, containing some of the information about the Java exception if possible.

exception Py4JSafeJavaError(*java_exception=None*, *str=None*)
Bases: Exception

exception DependencyCheckerError
Bases: Exception

exception JavaProcessError
Bases: Exception

Module contents

Tools for calling external (non-Python) tools.

modules

Subpackages

map

Submodules

benchmark

filter

multiproc

singleproc

threaded

Module contents

Submodules

base

This module provides base classes for Pimlico modules.

The procedure for creating a new module is the same whether you're contributing a module to the core set in the Pimlico codebase or a standalone module in your own codebase, or for a specific pipeline.

A Pimlico module is identified by the full Python-path to the Python package that contains it. This package should be laid out as follows:

- The module's metadata is defined by a class in info.py called ModuleInfo, which should inherit from BaseModuleInfo or one of its subclasses.
- The module's functionality is provided by a class in execute.py called ModuleExecutor, which should inherit from BaseModuleExecutor.

The exec Python module will not be imported until an instance of the module is to be run. This means that you can import dependencies and do any necessary initialization at the point where it's executed, without worrying about incurring the associated costs (and dependencies) every time a pipeline using the module is loaded.

class BaseModuleInfo(*module_name*, *pipeline*, *inputs={}*, *options={}*, *optional_outputs=[]*, *docstring=''*, *include_outputs=[]*, *alt_expanded_from=None*, *alt_param_settings=[]*, *module_variables={}*)

Bases: `object`

Abstract base class for all pipeline modules' metadata.

module_type_name = None

module_readable_name = None

module_options = {}

module_inputs = []
Specifies a list of (name, datatype instance) pairs for inputs that are always required

module_optional_inputs = []
Specifies a list of (name, datatype instance) pairs for optional inputs. The module's execution may vary depending on what is provided. If these are not given, None is returned from get_input()

module_optional_outputs = []
Specifies a list of (name, datatype instance) pairs for outputs that are written only if they're specified in the "output" option or used by another module

module_output_groups = []
List of output groups: (group_name, [output_name1, ...]). Further groups may be added by build_output_groups().

module_executable = True
Whether the module should be executed Typically True for almost all modules, except input modules (though some of them may also require execution) and filters

module_executor_override = None
If specified, this ModuleExecutor class will be used instead of looking one up in the exec Python module

main_module = None
Usually None. In the case of stages of a multi-stage module, stores a pointer to the main module.

module_supports_python2 = False
Most core Pimlico modules support use in Python 2 and 3. Modules that do should set this to True. If it is False, the module is assumed to work only in Python 3.

Since Python 2 compatibility requires extra work from the programmer, this is False by default.

To check whether a module can be used in Python 2, call `supports_python2()`, which will check this and also input and output datatypes.

module_outputs = []
Specifies a list of (name, datatype instance) pairs for outputs that are always written

classmethod supports_python2()

Returns True if the module can be run in Python 2 and 3, False if it only supports Python 3.

load_executor()
Loads a ModuleExecutor for this Pimlico module. Usually, this just involves calling *load_module_executor()*, but the default executor loading may be overridden for a particular module type by overriding this function. It should always return a subclass of ModuleExecutor, unless there's an error.

classmethod get_key_info_table()
When generating module docs, the table at the top of the page is produced by calling this method. It should return a list of two-item lists (title + value). Make sure to include the super-class call if you override this to add in extra module-specific info.

metadata_filename

get_metadata()

set_metadata_value(*attr*, *val*)

set_metadata_values(*val_dict*)

status

execution_history_path

add_execution_history_record(*line*)
Output a single line to the file that stores the history of module execution, so we can trace what we've done.

execution_history
Get the entire recorded execution history for this module. Returns an empty string if no history has been recorded.

input_names
All required inputs, first, then all supplied optional inputs

output_names

classmethod process_module_options(*opt_dict*)
Parse the options in a dictionary (probably from a config file), checking that they're valid for this model type.

Parameters **opt_dict** – dict of options, keyed by option name

Returns dict of options

classmethod extract_input_options(*opt_dict*, *module_name=None*, *previous_module_name=None*, *module_expansions={}*)
Given the config options for a module instance, pull out the ones that specify where the inputs come from and match them up with the appropriate input names.

The inputs returned are just names as they come from the config file. They are split into module name and output name, but they are not in any way matched up with the modules they connect to or type checked.

Parameters

- **module_name** – name of the module being processed, for error output. If not given, the name isn't included in the error.
- **previous_module_name** – name of the previous module in the order given in the config file, allowing a single-input module to default to connecting to this if the input connection wasn't given
- **module_expansions** – dictionary mapping module names to a list of expanded module names, where expansion has been performed as a result of alternatives in the parame-

ters. Provided here so that the unexpanded names may be used to refer to the whole list of module names, where a module takes multiple inputs on one input parameter

Returns dictionary of inputs

static choose_optional_outputs_from_options(*options*, *inputs*)
Normally, which optional outputs get produced by a module depend on the 'output' option given in the config file, plus any outputs that get used by subsequent modules. By overriding this method, module types can add extra outputs into the list of those to be included, conditional on other options.

It also receives the processed dictionary of inputs, so that the additional outputs can depend on what is fed into the input.

E.g. the corenlp module include the 'annotations' output if annotators are specified, so that the user doesn't need to give both options.

Note that this does not provide additional output definitions, just a list of the optional outputs (already defined) that should be included among the outputs produced.

static get_extra_outputs_from_options(*options*, *inputs*)
Normally, which optional outputs get produced by a module depend on the 'output' option given in the config file, plus any outputs that get used by subsequent modules. By overriding this method, module types can add extra outputs into the list of those to be included, conditional on other options.

It also receives the processed dictionary of inputs, so that the additional outputs can depend on what is fed into the input.

E.g. the corenlp module include the 'annotations' output if annotators are specified, so that the user doesn't need to give both options.

Note that this does not provide additional output definitions, just a list of the optional outputs (already defined) that should be included among the outputs produced.

provide_further_outputs()
Called during instantiation, once inputs and options are available, to add a further list of module outputs that are dependent on inputs or options.

When overriding this, you can provide a new docstring, which will be used in the module docs to describe the extra conditional outputs that are added.

build_output_groups()
Called during instantiation to produce a list of named groups of outputs. The list extends the statically define output groups in `module_output_groups`. You should use the static list unless you need to override this for conditionally added outputs.

Called after all input, options and output processing has been done, so the outputs in the attribute `available_outputs` are the final list of outputs that this module instance has.

Returns a list of groups, each specified as: `(group_name, [output_name1, ...])`.

May contain as many groups as necessary. They are not required to cover all the outputs and outputs may feature in multiple groups.

Should not include group "all", which is always included by default.

If you override this, use the docstring to specify what output groups will get added and how they are named. The text will be used in the generated module docs.

is_output_group_name(*group_name*)

get_output_group(*group_name*)
Get the list of output names corresponding to the given output group name.

Raises a KeyError if the output group does not exist.

get_module_output_dir(*absolute=False*, *short_term_store=None*)
Gets the path to the base output dir to be used by this module, relative to the storage base dir. When outputting data, the storage base dir will always be the short term store path, but when looking for the output data other base paths might be explored, including the long term store.

Kwarg `short_term_store` is included for backward compatibility, but outputs a deprecation warning.

Parameters **absolute** – if True, return absolute path to output dir in output store

Returns path, relative to store base path, or if absolute=True absolute path to output dir

get_absolute_output_dir(*output_name*)
The simplest way to get hold of the directory to use to output data to for a given output. This is the usual way to get an output directory for an output writer.

The directory is an absolute path to a location in the Pimlico output storage location.

Parameters **output_name** – the name of an output

Returns the absolute path to the output directory to use for the named output

get_output_dir(*output_name*, *absolute=False*, *short_term_store=None*)
Kwarg `short_term_store` is included for backward compatibility, but outputs a deprecation warning.

Parameters

- **absolute** – return an absolute path in the storage location used for output. If False (default), return a relative path, specified relative to the root of the Pimlico store used. This allows multiple stores to be searched for output
- **output_name** – the name of an output

Returns the path to the output directory to use for the named output, which may be relative to the root of the Pimlico store in use (default) or an absolute path in the output store, depending on *absolute*

get_output_datatype(*output_name=None*)
Get the datatype of a named output, or the default output. Returns an instance of the relevant PimlicoDatatype subclass. This can be used for typechecking and also for getting a reader for the output data, once it's ready, by supplying it with the path to the data.

To get a reader for the output data, use *get_output()*.

Parameters **output_name** – output whose datatype to retrieve. Default output if not specified

Returns

output_ready(*output_name=None*)
Check whether the named output is ready to be read from one of its possible storage locations.

Parameters **output_name** – output to check, or default output if not given

Returns False if data is not ready to be read

instantiate_output_reader_setup(*output_name*, *datatype*)
Produce a reader setup instance that will be used to prepare this reader. This provides functionality like checking that the data is ready to be read before the reader is instantiated.

The standard implementation uses the datatype's methods to get its standard reader setup and reader, but some modules may need to override this to provide other readers.

output_name is provided so that overriding methods' behaviour can be conditioned on which output is being fetched.

instantiate_output_reader(*output_name*, *datatype*, *pipeline*, *module=None*)
Prepare a reader for a particular output. The default implementation is very simple, but subclasses may override this for cases where the normal process of creating readers has to be modified.

Parameters

- **output_name** – output to produce a reader for
- **datatype** – the datatype for this output, already inferred

get_output_reader_setup(*output_name=None*)

get_output(*output_name=None*)
Get a reader corresponding to one of the outputs of the module. The reader will be that which corresponds to the output's declared datatype and will read the data from any of the possible locations where it can be found.

If the data is not available in any location, raises a `DataNotReadyError`.

To check whether the data is ready without calling this, call *output_ready()*.

get_output_writer(*output_name=None*, ***kwargs*)
Get a writer instance for the given output. Kwargs will be passed through to the writer and used to specify metadata and writer params.

Parameters

- **output_name** – output to get writer for, or default output if left
- **kwargs** –

Returns

is_multiple_input(*input_name=None*)
Returns True if the named input (or default input if no name is given) is a MultipleInputs input, False otherwise. If it is, get_input() will return a list, otherwise it will return a single datatype.

get_input_module_connection(*input_name=None*, *always_list=False*)
Get the ModuleInfo instance and output name for the output that connects up with a named input (or the first input) on this module instance. Used by get_input() – most of the time you probably want to use that to get the instantiated datatype for an input.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single (module, output_name) pair. If always_list=True, in this latter case we return a single-item list.

get_input_datatype(*input_name=None*, *always_list=False*)
Get a list of datatype instances corresponding to one of the inputs to the module. If an input name is not given, the first input is returned.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single datatype.

get_input_reader_setup(*input_name=None*, *always_list=False*)
Get reader setup for one of the inputs to the module. Looks up the corresponding output from another module and uses that module's metadata to get that output's instance. If an input name is not given, the first input is returned.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single datatype instance. If always_list=True, in this latter case we return a single-item list.

If the requested input name is an optional input and it has not been supplied, returns None.

You can get a reader for the input, once the data is ready to be read, by calling *get_reader()* on the setup object. Or use *get_input()* on the module.

get_input(*input_name=None*, *always_list=False*)
Get a reader for one of the inputs to the module. Should only be called once the input data is ready to read. It's therefore fine to call this from a module executor, since data availability has already been checked by this point.

If the input type was specified with `MultipleInputs`, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single datatype instance. If `always_list=True`, in this latter case we return a single-item list.

If the requested input name is an optional input and it has not been supplied, returns None.

Similarly, if you run in preliminary mode, multiple inputs might produce None for some of their inputs if the data is not ready.

input_ready(*input_name=None*)
Check whether the data is ready to go corresponding to the named input.

Parameters **input_name** – input to check

Returns True if input is ready

all_inputs_ready()
Check *input_ready()* on all inputs.

Returns True if all input datatypes are ready to be used

classmethod is_filter()

missing_module_data()
Reports missing data not associated with an input dataset.

Calling *missing_data()* reports any problems with input data associated with a particular input to this module. However, modules may also rely on data that does not come from one of their inputs. This happens primarily (perhaps solely) when a module option points to a data source. This might be the case with any module, but is particularly common among input reader modules, which have no inputs, but read data according to their options.

Returns list of problems

missing_data(*input_names=None*, *assume_executed=[]*, *assume_failed=[]*, *allow_preliminary=False*)
Check whether all the input data for this module is available. If not, return a list strings indicating which outputs of which modules are not available. If it's all ready, returns an empty list.

To check specific inputs, give a list of input names. To check all inputs, don't specify *input_names*. To check the default input, give *input_names=[None]*. If not checking a specific input, also checks non-input data (see *missing_module_data()*).

If *assume_executed* is given, it should be a list of module names which may be assumed to have been executed at the point when this module is executed. Any outputs from those modules will be excluded from the input checks for this module, on the assumption that they will have become available, even if they're not currently available, by the time they're needed.

If *assume_executed* is given, it should be a list of module names which should be assumed to have failed. If we rely on data from the output of one of them, instead of checking whether it's available we simply assume it's not.

Why do this? When running multiple modules in sequence, if one fails it is possible that its output datasets look like complete datasets. For example, a partially written iterable corpus may look like a perfectly valid corpus, which happens to be smaller than it should be. After the execution failure, we may check other

modules to see whether it's possible to run them. Then we need to know not to trust the output data from the failed module, even if it looks valid.

If *allow_preliminary=True*, for any inputs that are multiple inputs and have multiple connections to previous modules, consider them to be satisfied if at least one of their inputs is ready. The normal behaviour is to require all of them to be ready, but in a preliminary run this requirement is relaxed.

classmethod is_input()

dependencies

Returns list of names of modules that this one depends on for its inputs.

get_transitive_dependencies()
Transitive closure of *dependencies*.

Returns list of names of modules that this one recursively (transitively) depends on for its inputs.

typecheck_inputs()

typecheck_input(*input_name*)
Typecheck a single input. `typecheck_inputs()` calls this and is used for typechecking of a pipeline. This method returns the (or the first) satisfied input requirement, or raises an exception if typechecking failed, so can be handy separately to establish which requirement was met.

The result is always a list, but will contain only one item unless the input is a multiple input.

get_software_dependencies()
Check that all software required to execute this module is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

get_input_software_dependencies()
Collects library dependencies from the input datatypes to this module, which will need to be satisfied for the module to be run.

Unlike *get_software_dependencies()*, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

get_output_software_dependencies()
Collects library dependencies from the output datatypes to this module, which will need to be satisfied for the module to be run.

Unlike *get_input_software_dependencies()*, it may not be the case that all of these dependencies strictly need to be satisfied before the module can be run. It could be that a datatype can be written without satisfying all the dependencies needed to read it. However, we assume that dependencies of all output datatypes must be satisfied in order to run the module that writes them, since this is usually the case, and these are checked before running the module.

Unlike *get_software_dependencies()*, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

check_ready_to_run()
Called before a module is run, or if the 'check' command is called. This will only be called after all library dependencies have been confirmed ready (see :method:get_software_dependencies).

Essentially, this covers any module-specific checks that used to be in check_runtime_dependencies() other than library installation (e.g. checking models exist).

Always call the super class' method if you override.

Returns a list of (name, description) pairs, where the name identifies the problem briefly and the description explains what's missing and (ideally) how to fix it.

reset_execution()
Remove all output data and metadata from this module to make a fresh start, as if it's never been executed.

May be overridden if a module has some side effect other than creating/modifying things in its output directory(/ies), but overridden methods should always call the super method. Occasionally this is necessary, but most of the time the base implementation is enough.

get_detailed_status()
Returns a list of strings, containing detailed information about the module's status that is specific to the module type. This may include module-specific information about execution status, for example.

Subclasses may override this to supply useful (human-readable) information specific to the module type. They should called the super method.

classmethod module_package_name()
The package name for the module, which is used to identify it in config files. This is the package containing the info.py in which the ModuleInfo is defined.

get_execution_dependency_tree()
Tree of modules that will be executed when this one is executed. Where this module depends on filters, the tree goes back through them to find what they depend on (since they will be executed simultaneously)

get_all_executed_modules()
Returns a list of all the modules that will be executed when this one is (including itself). This is the current module (if executable), plus any filters used to produce its inputs.

lock_path

lock()
Mark the module as locked, so that it cannot be executed. Called when execution begins, to ensure that you don't end up executing the same module twice simultaneously.

unlock()
Remove the execution lock on this module.

is_locked()

Returns True is the module is currently locked from execution

get_log_filenames(*name='error'*)
Get a list of all the log filenames of the given prefix that exist in the module's output dir. They will be ordered according to their numerical suffixes (i.e. the order in which they were created).

Returns a list of (filename, num) tuples, where num is the numerical suffix as an int.

get_new_log_filename(*name='error'*)
Returns an absolute path that can be used to output a log file for this module. This is used for outputting error logs. It will always return a filename that doesn't currently exist, so can be used multiple times to output multiple logs.

get_last_log_filename(*name='error'*)
Get the most recent error log that was created by a call to get_new_log_filename(). Returns an absolute path, or None if no matching files are found.

collect_unexecuted_dependencies(*modules*)
Given a list of modules, checks through all the modules that they depend on to put together a list of modules that need to be executed so that the given list will be left in an executed state. The list includes the modules themselves, if they're not fully executed, and unexecuted dependencies of any unexecuted modules (recursively).

Parameters **modules** – list of ModuleInfo instances

Returns list of ModuleInfo instances that need to be executed

collect_runnable_modules(*pipeline*, *preliminary=False*)
Look for all unexecuted modules in the pipeline to find any that are ready to be executed. Keep collecting runnable modules, including those that will become runnable once we've run earlier ones in the list, to produce a list of a sequence of modules that could be set running now.

Parameters **pipeline** – pipeline config

Returns ordered list of runable modules. Note that it must be run in this order, as some might depend on earlier ones in the list

satisfies_typecheck(*provided_type*, *type_requirements*)
Interface to Pimlico's standard type checking (see *check_type*) that returns a boolean to say whether type checking succeeded or not.

check_type(*provided_type*, *type_requirements*)
Type-checking algorithm for making sure outputs from modules connect up with inputs that they satisfy the requirements for.

type_checking_name(*typ*)

class BaseModuleExecutor(*module_instance_info*, *stage=None*, *debug=False*, *force_rerun=False*)
Bases: `object`

Abstract base class for executors for Pimlico modules. These are classes that actually do the work of executing the module on given inputs, writing to given output locations.

execute()
Run the actual module execution.

May return None, in which case it's assumed to have fully completed. If a string is returned, it's used as an alternative module execution status. Used, e.g., by multi-stage modules that need to be run multiple times.

exception ModuleInfoLoadError(**args*, ***kwargs*)
Bases: `Exception`

exception ModuleExecutorLoadError
Bases: `Exception`

exception ModuleTypeError
Bases: `Exception`

exception TypeCheckError(**args*, ***kwargs*)
Bases: `Exception`

Pipeline type-check mismatch.

Full description of problem provided in error message. May optionally provide more detailed information about the input and output (source) that failed to match, the expected type and the received type, all as strings. Specify using kwargs `input`, `source`, `required_type` and `provided_type`.

format()
Provide a nice visual format of the mismatch to help the user.

exception DependencyError(*message*, *stderr=None*, *stdout=None*)
Bases: `Exception`

Raised when a module's dependencies are not satisfied. Generally, this means a dependency library needs to be installed, either on the local system or (more often) by calling the appropriate make target in the lib directory.

load_module_executor(*path_or_info*)
Utility for loading the executor class for a module from its full path. More or less just a wrapper around an import, with some error checking. Locates the executor by a standard procedure that involves checking for an "execute" python module alongside the info's module.

Note that you shouldn't generally use this directly, but instead call the *load_executor()* method on a module info (which will call this, unless special behaviour has been defined).

Parameters **path** – path to Python package containing the module

Returns class

load_module_info(*path*)
Utility to load the metadata for a Pimlico pipeline module from its package Python path.

Parameters **path** –

Returns

execute

Runtime execution of modules

This module provides the functionality to check that Pimlico modules are ready to execute and execute them. It is used by the *run* command.

check_and_execute_modules(*pipeline*, *module_names*, *force_rerun=False*, *debug=False*, *log=None*, *all_deps=False*, *check_only=False*, *exit_on_error=False*, *preliminary=False*, *email=None*)
Main method called by the *run* command that first checks a pipeline, checks all pre-execution requirements of the modules to be executed and then executes each of them. The most common case is to execute just one module, but a sequence may be given.

Parameters

- **exit_on_error** – drop out if a ModuleExecutionError occurs in any individual module, instead of continuing to the next module that can be run
- **pipeline** – loaded PipelineConfig
- **module_names** – list of names of modules to execute in the order they should be run
- **force_rerun** – execute modules, even if they're already marked as complete
- **debug** – output debugging info
- **log** – logger, if you have one you want to reuse
- **all_deps** – also include unexecuted dependencies of the given modules
- **check_only** – run all checks, but stop before executing. Used for *check* command

Returns

check_modules_ready(*pipeline*, *modules*, *log*, *preliminary=False*)
Check that a module is ready to be executed. Always called before execution begins.

Parameters

- **pipeline** – loaded PipelineConfig
- **modules** – loaded ModuleInfo instances, given in the order they're going to be executed. For each module, it's assumed that those before it in the list have already been run when it is run.
- **log** – logger to output to

Returns If *preliminary=True*, list of problems that were ignored by allowing preliminary run. Otherwise, None – we raise an exception when we first encounter a problem

execute_modules(*pipeline*, *modules*, *log*, *force_rerun=False*, *debug=False*, *exit_on_error=False*, *preliminary=False*, *email=None*)

format_execution_dependency_tree(*tree*)
Takes a tree structure of modules and their inputs, tracing where inputs to a module come from, and formats it recursively for output to the logs.

Parameters **tree** – pair (module name, inputs list), where each input is a tuple (input name, previous module output name, previous module subtree)

Returns list of lines of formatted string

send_final_report_email(*pipeline*, *error_modules*, *success_modules*, *skipped_modules*, *all_modules*)

send_module_report_email(*pipeline*, *module*, *short_error*, *long_error*)

exception ModuleExecutionError(**args*, ***kwargs*)
Bases: `Exception`

Base for any errors encountered during execution of a module.

Note that the `cause` attribute is used to trace the cause of an exception, so a chain can be built.

This is now provided as standard using the `raise ... from ...` syntax in Python 3, which can be accessed in Python 2 using future's `raise_from()`. The `cause` attribute should gradually be replaced by this, which works better.

exception ModuleNotReadyError(**args*, ***kwargs*)
Bases: *pimlico.core.modules.execute.ModuleExecutionError*

exception ModuleAlreadyCompletedError(**args*, ***kwargs*)
Bases: *pimlico.core.modules.execute.ModuleExecutionError*

exception StopProcessing
Bases: `Exception`

inputs

multistage

class MultistageModuleInfo(*module_name*, *pipeline*, ***kwargs*)
Bases: *pimlico.core.modules.base.BaseModuleInfo*

Base class for multi-stage modules. You almost certainly don't want to override this yourself, but use the factory method instead. It exists mainly for providing a way of identifying multi-stage modules.

module_executable = True

stages = None

typecheck_inputs()
Overridden to check internal output-input connections as well as the main module's inputs.

get_software_dependencies()
Check that all software required to execute this module is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

get_input_software_dependencies()
Collects library dependencies from the input datatypes to this module, which will need to be satisfied for the module to be run.

Unlike *get_software_dependencies()*, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

check_ready_to_run()
Called before a module is run, or if the 'check' command is called. This will only be called after all library dependencies have been confirmed ready (see :method:get_software_dependencies).

Essentially, this covers any module-specific checks that used to be in check_runtime_dependencies() other than library installation (e.g. checking models exist).

Always call the super class' method if you override.

Returns a list of (name, description) pairs, where the name identifies the problem briefly and the description explains what's missing and (ideally) how to fix it.

get_detailed_status()
Returns a list of strings, containing detailed information about the module's status that is specific to the module type. This may include module-specific information about execution status, for example.

Subclasses may override this to supply useful (human-readable) information specific to the module type. They should called the super method.

reset_execution()
Remove all output data and metadata from this module to make a fresh start, as if it's never been executed.

May be overridden if a module has some side effect other than creating/modifying things in its output directory(/ies), but overridden methods should always call the super method. Occasionally this is necessary, but most of the time the base implementation is enough.

classmethod get_key_info_table()
Add the stages into the key info table.

get_next_stage()
If there are more stages to be executed, returns a pair of the module info and stage definition. Otherwise, returns (None, None)

status

is_locked()

Returns True is the module is currently locked from execution

multistage_module(*multistage_module_type_name*, *module_stages*, *use_stage_option_names=False*, *module_readable_name=None*)

Factory to build a multi-stage module type out of a series of stages, each of which specifies a module type for the stage. The stages should be a list of *ModuleStage* objects.

class ModuleStage(*name*, *module_info_cls*, *connections=None*, *output_connections=None*, *option_connections=None*, *use_stage_option_names=False*, *extra_connections_from_options=None*)

Bases: object

A single stage in a multi-stage module.

If no explicit input connections are given, the default input to this module is connected to the default output from the previous.

Connections can be given as a list of ModuleConnection s.

Output connections specify that one of this module's outputs should be used as an output from the multi-stage module. Optional outputs for the multi-stage module are not currently supported (though could in theory be added later). This should be a list of ModuleOutputConnection s. If none are given for any of the stages, the module will have a single output, which is the default output from the last stage.

Option connections allow you to specify the names that are used for the multistage module's options that get passed through to this stage's module options. Simply specify a dict for option_connections where the keys are names module options for this stage and the values are the names that should be used for the multistage module's options.

You may map multiple options from different stages to the same option name for the multistage module. This will result in the same option value being passed through to both stages. Note that help text, option type, option processing, etc will be taken from the first stage's option (in case the two options aren't identical).

Options not explicitly mapped to a name will use the name <stage_name>_<option_name>. If use_stage_option_names=True, this prefix will not be added: the stage's option names will be used directly as the option name of the multistage module. Note that there is a danger of clashing option names with this behaviour, so only do it if you know the stages have distinct option names (or should share their values where the names overlap).

Further connections may be produced once processed options are available (when the main module's module info is instantiated), by specifying a one-argument function as extra_connections_from_options. The argument is the processed option dictionary, which will contain the full set of options given the to the main module.

class ModuleConnection

Bases: object

class InternalModuleConnection(*input_name*, *output_name=None*, *previous_module=None*)

Bases: *pimlico.core.modules.multistage.ModuleConnection*

Connection between the output of one module in the multi-stage module and the input to another.

May specify the name of the previous module that a connection should be made to. If this is not given, the previous module in the sequence will be assumed.

If *output_name=None*, connects to the default output of the previous module.

class InternalModuleMultipleConnection(*input_name*, *outputs*)

Bases: *pimlico.core.modules.multistage.ModuleConnection*

Connection between the outputs of multiple modules and the input to another (which must be a multiple input).

`outputs` should be a list of (`module_name`, `output_name`) pairs, or just strings giving the output name, assumed to be from the previous module.

class ModuleInputConnection(*stage_input_name=None*, *main_input_name=None*)
Bases: *pimlico.core.modules.multistage.ModuleConnection*

Connection of a sub-module's input to an input to the multi-stage module.

If *main_input_name* is not given, the name for the input to the multistage module will be identical to the stage input name. This might lead to unintended behaviour if multiple inputs end up with the same name, so you can specify a different name if necessary to avoid clashes.

If multiple inputs (e.g. from different stages) are connected to the same main input name, they will take input from the same previous module output. Nothing clever is done to unify the type requirements, however: the first stage's type requirement is used for the main module's input.

If *stage_input_name* is not given, the module's default input will be connected.

class ModuleOutputConnection(*stage_output_name=None*, *main_output_name=None*)
Bases: `object`

Specifies the connection of a sub-module's output to the multi-stage module's output. Works in a similar way to *ModuleInputConnection*.

exception MultistageModulePreparationError
Bases: `Exception`

options

Utilities and type processors for module options.

opt_type_help(*help_text*)
Decorator to add help text to functions that are designed to be used as module option processors. The help text will be used to describe the type in documentation.

opt_type_example(*example_text*)
Decorate to add an example value to function that are designed to be used as module option processors. The given text will be used in module docs as an example of how to specify the option in a config file.

format_option_type(*t*)

str_to_bool(*string*)
Convert a string value to a boolean in a sensible way. Suitable for specifying booleans as options.

Parameters **string** – input string

Returns boolean value

choose_from_list(*options*, *name=None*)
Utility for option processors to limit the valid values to a list of possibilities.

comma_separated_list(*item_type=<class 'str'>*, *length=None*)
Option processor type that accepts comma-separated lists of strings. Each value is then parsed according to the given item_type (default: string).

comma_separated_strings(*string*)

json_string(*string*)

json_dict(*string*)
JSON dicts, with or without {}s

process_module_options(*opt_def*, *opt_dict*, *module_type_name*)
Utility for processing runtime module options. Called from module base class.

Also used when loading a dataset's datatype from datatype options specified in a config file.

Parameters

- **opt_def** – dictionary defining available options
- **opt_dict** – dictionary of option values
- **module_type_name** – name for error output

Returns dictionary of processed options

exception ModuleOptionParseError
Bases: `Exception`

Module contents

Core functionality for loading and executing different types of pipeline module.

Submodules

config

Reading of pipeline config from a file into the data structure used to run and manipulate the pipeline's data.

class PipelineConfig(*name*, *pipeline_config*, *local_config*, *filename=None*, *variant='main'*, *available_variants=[]*, *log=None*, *all_filenames=None*, *module_aliases={}*, *local_config_sources=None*, *section_headings=None*)
Bases: `object`

Main configuration for a pipeline, read in from a config file.

For details on how to write config files that get read by this class, see *Pipeline config*.

modules
List of module names, in the order they were specified in the config file.

module_dependencies
Dictionary mapping a module name to a list of the names of modules that it depends on for its inputs.

module_dependents
Opposite of module_dependencies. Returns a mapping from module names to a list of modules the depend on the module.

get_dependent_modules(*module_name*, *recurse=False*, *exclude=[]*)
Return a list of the names of modules that depend on the named module for their inputs.

If *exclude* is given, we don't perform a recursive call on any of the modules in the list. For each item we recurse on, we extend the exclude list in the recursive call to include everything found so far (in other recursive calls). This avoids unnecessary recursion in complex pipelines.

If *exclude=None*, it is also passed through to recursive calls as None. Its default value of *[]* avoids excessive recursion from the top-level call, by allowing things to be added to the exclusion list for recursive calls.

Parameters **recurse** – include all transitive dependents, not just those that immediately depend on the module.

append_module(*module_info*)
Add a moduleinfo to the end of the pipeline. This is mainly for use while loaded a pipeline from a config file.

get_module_schedule()
Work out the order in which modules should be executed. This is an ordering that respects dependencies, so that modules are executed after their dependencies, but otherwise follows the order in which modules were specified in the config.

Returns list of module names

reset_all_modules()
Resets the execution states of all modules, restoring the output dirs as if nothing's been run.

path_relative_to_config(*path*)
Get an absolute path to a file/directory that's been specified relative to a config file (usually within the config file).

If the path is already an absolute path, doesn't do anything.

Parameters **path** – relative path

Returns absolute path

short_term_store
For backwards compatibility: returns output path

long_term_store
For backwards compatibility: return storage location 'long' if it exists, else first storage location

named_storage_locations

store_names

output_path

static load(*filename*, *local_config=None*, *variant='main'*, *override_local_config={}*, *only_override_config=False*)
Main function that loads a pipeline from a config file.

Parameters

- **filename** – file to read config from
- **local_config** – location of local config file, where we'll read system-wide config. Usually not specified, in which case standard locations are searched. When loading programmatically, you might want to give this
- **variant** – pipeline variant to load
- **override_local_config** – extra configuration values to override the system-wide config
- **only_override_config** – don't load local config from files, just use that given in override_local_config. Used for loading test pipelines

Returns

static load_local_config(*filename=None*, *override={}*, *only_override=False*)
Load local config parameters. These are usually specified in a *.pimlico* file, but may be overridden by other config locations, on the command line, or elsewhere programmatically.

If only_override=True, don't load any files, just use the values given in override. The various locations for local config files will not be checked (which usually happens when filename=None). This is not useful for normal pipeline loading, but is used for loading test pipelines.

static trace_load_local_config (*filename=None*, *override={}*, *only_override=False*)
Trace the process of loading local config file(s). Follows exactly the same logic as load_local_config(), but documents what it finds/doesn't find.

static empty (*local_config=None*, *override_local_config={}*, *override_pipeline_config={}*, *only_override_config=False*)
Used to programmatically create an empty pipeline. It will contain no modules, but provides a gateway to system info, etc and can be used in place of a real Pimlico pipeline.

Parameters

- **local_config** – filename to load local config from. If not given, the default locations are searched
- **override_local_config** – manually override certain local config parameters. Dict of parameter values
- **only_override_config** – don't load any files, just use the values given in override. The various locations for local config files will not be checked (which usually happens when filename=None). This is not useful for normal pipeline loading, but is used for loading test pipelines.

Returns the *PipelineConfig* instance

find_data_path (*path*, *default=None*)
Given a path to a data dir/file relative to a data store, tries taking it relative to various store base dirs. If it exists in a store, that absolute path is returned. If it exists in no store, return None. If the path is already an absolute path, nothing is done to it.

Searches all the specified storage locations.

Parameters

- **path** – path to data, relative to store base
- **default** – usually, return None if no data is found. If default is given, return the path relative to the named storage location if no data is found. Special value "output" returns path relative to output location, whichever of the storage locations that might be

Returns absolute path to data, or None if not found in any store

find_data_store (*path*, *default=None*)
Like *find_data_path()*, searches through storage locations to see if any of them include the data that lives at this relative path. This method returns the name of the store in which it was found.

Parameters

- **path** – path to data, relative to store base
- **default** – usually, return None if no data is found. If default is given, return the path relative to the named storage location if no data is found. Special value "output" returns path relative to output location, whichever of the storage locations that might be

Returns name of store

find_data (*path*, *default=None*)
Given a path to a data dir/file relative to a data store, tries taking it relative to various store base dirs. If it exists in a store, that absolute path is returned. If it exists in no store, return None. If the path is already an absolute path, nothing is done to it.

Searches all the specified storage locations.

Parameters

- **path** – path to data, relative to store base

- **default** – usually, return None if no data is found. If default is given, return the path relative to the named storage location if no data is found. Special value "output" returns path relative to output location, whichever of the storage locations that might be

Returns (store, path), where store is the name of the store used and path is absolute path to data, or None for both if not found in any store

get_data_search_paths(*path*)
Like *find_all_data_paths()*, but returns a list of all absolute paths which this data path could correspond to, whether or not they exist.

Parameters **path** – relative path within Pimlico directory structures

Returns list of string

step

enable_step()
Enable super-verbose, interactive step mode.

::seealso:

```
Module :mod:pimlico.cli.debug
   The debug module defines the behaviour of step mode.
```

exception PipelineConfigParseError(**args*, ***kwargs*)
Bases: Exception

General problems interpreting pipeline config

exception PipelineStructureError(**args*, ***kwargs*)
Bases: Exception

Fundamental structural problems in a pipeline.

exception PipelineCheckError(*cause*, **args*, ***kwargs*)
Bases: Exception

Error in the process of explicitly checking a pipeline for problems.

preprocess_config_file(*filename*, *variant='main'*, *initial_vars={}*)
Workhorse of the initial part of config file reading. Deals with all of our custom stuff for pipeline configs, such as preprocessing directives and includes.

Parameters

- **filename** – file from which to read main config
- **variant** – name of a variant to load. The default (*main*) loads the main variant, which always exists
- **initial_vars** – variable assignments to make available for substitution. This will be added to by any *vars* sections that are read.

Returns tuple: raw config dict; list of variants that could be loaded; final vars dict; list of filenames that were read, including included files; dict of docstrings for each config section

check_for_cycles(*pipeline*)
Basic cyclical dependency check, always run on pipeline before use.

check_release(*release_str*)
Check a release name against the current version of Pimlico to determine whether we meet the requirement.

check_pipeline(*pipeline*)
Checks a pipeline over for metadata errors, cycles, module typing errors and other problems. Called every time a pipeline is loaded, to check the whole pipeline's metadata is in order.

Raises a *PipelineCheckError* if anything's wrong.

get_dependencies(*pipeline*, *modules*, *recursive=False*, *sources=False*)
Get a list of software dependencies required by the subset of modules given.

If recursive=True, dependencies' dependencies are added to the list too.

Parameters

- **pipeline** –
- **modules** – list of modules to check. If None, checks all modules

print_missing_dependencies(*pipeline*, *modules*)
Check runtime dependencies for a subset of modules and output a table of missing dependencies.

Parameters

- **pipeline** –
- **modules** – list of modules to check. If None, checks all modules

Returns True if no missing dependencies, False otherwise

print_dependency_leaf_problems(*dep*, *local_config*)

logs

get_log_file(*name*)
Returns the path to a log file that may be used to output helpful logging info. Typically used to output verbose error information if something goes wrong. The file can be found in the Pimlico log dir.

Parameters **name** – identifier to distinguish from other logs

Returns path

paths

abs_path_or_model_dir_path(*path*, *model_type*)

Module contents

datatypes

Subpackages

corpora

Subpackages

parse

Submodules

trees

Datatypes for storing parse trees from constitutency parsers.

Note: Parse tress are temporary implementations that don't actually parse the data, but just split it into sentences. That is, they store the raw output from the OpenNLP parser. In future, this should be replaced by a generic tree structure storage.

---

class OpenNLPTreeStringsDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

The attribute `trees` provides a list of strings representing each of the trees in the document, usually one per sentence.

Todo: In future, this should be replaced by a doc type that reads in the parse trees and returns a tree data structure. For now, you need to load and process the tree strings yourself.

---

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for OpenNLPTreeStringsDocumentType

keys = ['trees']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

Module contents

Submodules

base

class CountInvalidCmd
Bases: *pimlico.cli.shell.base.ShellCommand*

Data shell command to count up the number of invalid docs in a tarred corpus. Applies to any iterable corpus.

commands = ['invalid']

help_text = 'Count the number of invalid documents in this dataset'

execute (*shell*, **args*, ***kwargs*)
Execute the command. Get the dataset reader as shell.data.

Parameters

- **shell** – DataShell instance. Reader available as shell.data
- **args** – Args given by the user
- **kwargs** – Named args given by the user as key=val

data_point_type_opt (*text*)

class IterableCorpus (**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Superclass of all datatypes which represent a dataset that can be iterated over document by document (or datapoint by datapoint - what exactly we're iterating over may vary, though documents are most common).

This is an abstract base class and doesn't provide any mechanisms for storing documents or organising them on disk in any way. Many input modules will override this to provide a reader that iterates over the documents directly, according to IterableCorpus' interface. The main subclass of this used within pipelines is GroupedCorpus, which provides an interface for iterating over groups of documents and a storage mechanism for grouping together documents in archives on disk.

May be used as a type requirement, but remember that it is not possible to create a reader from this type directly: use a subtype, like `GroupedCorpus`, instead.

The actual type of the data depends on the type given as the first argument, which should be an instance of DataPointType or a subclass: it could be, e.g. coref output, etc. Information about the type of individual documents is provided by *data_point_type* and this is used in type checking.

Note that the data point type is the first datatype option, so can be given as the first positional arg when instantiating an iterable corpus subtype:

```
corpus_type = GroupedCorpus(RawTextDocumentType())
corpus_reader = corpus_type("... base dir path ...")
```

At creation time, length should be provided in the metadata, denoting how many documents are in the dataset.

datatype_name = 'iterable_corpus'

shell_commands = [<pimlico.datatypes.corpora.base.CountInvalidCmd object>]

datatype_options = {'data_point_type': {'default': DataPointType(), 'help': 'Data p

datatype_supports_python2 = True

supports_python2 ()
Whether a corpus type supports Python 2, depends on its document type. The corpus datatype introduces no reason not to, but specific document types might.

run_browser (*reader*, *opts*)
Launches a browser interface for reading this datatype, browsing the data provided by the given reader.

Not all datatypes provide a browser. For those that don't, this method should raise a NotImplementedError.

opts provides the argparser options from the command line.

This tool used to be only available for iterable corpora, but now it's possible for any datatype to provide a browser. IterableCorpus provides its own browser, as before, which uses one of the data point type's formatters to format documents.

class Reader(**args*, ***kwargs*)

Bases: pimlico.datatypes.base.Reader

Reader class for IterableCorpus

get_detailed_status()

Returns a list of strings, containing detailed information about the data.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

list_iter()

Iterate over the list of document names, without yielding the doc contents.

Whilst this could be considerably faster than iterating over all the docs, the default implementation, if not overridden by subclasses of IterableCorpus, simply calls the doc iter and throws away the docs.

data_to_document(*data*, *metadata=None*)

Applies the corpus' datatype's processing to the raw data, given as a bytes object, and produces a document instance.

Parameters

- **metadata** – dict containing doc metadata (optional)
- **data** – bytes raw data

Returns document instance

class Setup(*datatype*, *data_paths*)

Bases: pimlico.datatypes.base.Setup

Setup class for IterableCorpus.Reader

data_ready(*path*)

Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()

Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()

Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)

Instantiate a reader using this setup.

Parameters

- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()

May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type
alias of *IterableCorpus.Reader*

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.
Returns True if the reader's ready to be instantiated, False otherwise

class Writer(*datatype*, **args*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Stores the length of the corpus.

NB: IterableCorpus itself has no particular way of storing files, so this is only here to ensure that all subclasses (e.g. GroupedCorpus) store a length in the same way.

metadata_defaults = {'length': (None, 'Number of documents in the corpus. Must be

writer_param_defaults = {}

check_type(*supplied_type*)
Override type checking to require that the supplied type have a document type that is compatible with (i.e. a subclass of) the document type of this class.

The data point types can also introduce their own checks, other than simple isinstance checks.

type_checking_name()
Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

full_datatype_name()
Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

data_points

Document types used to represent datatypes of individual documents in an IterableCorpus or subtype.

class DataPointType(**args*, ***kwargs*)
Bases: object

Base data-point type for iterable corpora. All iterable corpora should have data-point types that are subclasses of this.

Every data point type has a corresponding document class, which can be accessed as *MyDataPointType.Document*. When overriding data point types, you can define a nested *Document* class, with no base class, to override parts of the document class' functionality or add new methods, etc. This will be used to automatically create the *Document* class for the data point type.

Some data-point types may specify some options, using the data_point_type_options field. This works in the same way as PimlicoDatatype's datatype_options. Values for the options can be specified on initialization as args or kwargs of the data-point type.

Note: I have now implemented the data-point type options, just like datatype options. However, you cannot yet specify these in a config file when loading a stored corpus. An additional datatype option should be added to iterable corpora that allows you to specify data point type options for when a datatype is being loaded using a config file.

formatters = []
List of (name, cls_path) pairs specifying a standard set of formatters that the user might want to choose from to view a dataset of this type. The user is not restricted to this set, but can easily choose these by name, instead of specifying a class path themselves. The first in the list is the default used if no formatter is specified. Falls back to DefaultFormatter if empty

metadata_defaults = {}
Metadata keys that should be written for this data point type, with default values and strings documenting the meaning of the parameter. Used for writers for this data point type. See `Writer`.

data_point_type_options = {}
Options specified in the same way as module options that control the nature of the document type. These are not things to do with reading of specific datasets, for which the dataset's metadata should be used. These are things that have an impact on typechecking, such that options on the two checked datatypes are required to match for the datatypes to be considered compatible.

This corresponds exactly to a PimlicoDatatype's datatype_options and is processed in the same way.

They should always be an ordered dict, so that they can be specified using positional arguments as well as kwargs and config parameters.

data_point_type_supports_python2 = True
Most core Pimlico datatypes support use in Python 2 and 3. Datatypes that do should set this to True. If it is False, the datatype is assumed to work only in Python 3.

Python 2 compatibility requires extra work from the programmer. Datatypes should generally declare whether or not they provide this support by overriding this explicitly.

Use `supports_python2()` to check whether a data-point type instance supports Python 2. (There may be reasons for a datatype's instance to override this class-level setting.)

supports_python2()
Just returns data_point_type_supports_python2.

name

check_type(*supplied_type*)
Type checking for an iterable corpus calls this to check that the supplied data point type matches the required one (i.e. this instance). By default, the supplied type is simply required to be an instance of the required type (or one of its subclasses).

This may be overridden to introduce other type checks.

is_type_for_doc(*doc*)
Check whether the given document is of this type, or a subclass of this one.

If the object is not a document instance (or, more precisely, doesn't have a data_point_type attr), this will always return False.

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

classmethod full_class_name()
The fully qualified name of the class for this data point type, by which it is referenced in config files. Used in docs

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: `object`

The abstract superclass of all documents.

You do not need to subclass or instantiate these yourself: subclasses are created automatically to correspond to each document type. You can add functionality to a datapoint type's document by creating a nested *Document* class. This will inherit from the parent datapoint type's document. This happens automatically - you don't need to do it yourself and shouldn't inherit from anything:

```
class MyDataPointType(DataPointType):
    class Document:
        # Overide document things here
        # Add your own methods, properties, etc for getting data from the␣
↪document
```

A data point type's constructed document class is available as *MyDataPointType.Document*.

Each document type should provide a method to convert from raw data (a bytes object in Py3, or `future`'s backport of `bytes` in Py2) to the internal representation (an arbitrary dictionary) called *raw_to_internal()*, and another to convert the other way called *internal_to_raw()*. Both forms of the data are available using the properties *raw_data* and *internal_data*, and these methods are called as necessary to convert back and forth.

This is to avoid unnecessary conversions. For example, if the raw data is supplied and then only the raw data is ever used (e.g. passing the document straight through and writing out to disk), we want to avoid converting back and forth.

A subtype should then supply methods or properties (typically using the cached_property decorator) to provide access to different parts of the data. See the many built-in document types for examples of doing this.

You should not generally need to override the *__init__* method. You may, however, wish to override *internal_available()* or *raw_available()*. These are called as soon as the internal data or raw data, respectively, become available, which may be at instantiation or after conversion. This can be useful if there are bits of computation that you want to do on the basis of one of these and then store to avoid repeated computation.

keys = []
Specifies the keys that a document has in its internal data Subclasses should specify their keys The internal data fields corresponding to these can be accessed as attributes of the document

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

raw_available()
Called as soon as the raw data becomes available, either at instantiation or conversion.

internal_available()
Called as soon as the internal data becomes available, either at instantiation or conversion.

raw_data

internal_data

class InvalidDocument(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.DataPointType*

Widely used in Pimlico to represent an empty document that is empty not because the original input document was empty, but because a module along the way had an error processing it. Document readers/writers should generally be robust to this and simply pass through the whole thing where possible, so that it's always possible to work out, where one of these pops up, where the error occurred.

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for InvalidDocument

keys = ['module_name', 'error_info']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

module_name

error_info

class RawDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.DataPointType*

Base document type. All document types for grouped corpora should be subclasses of this.

It may be used itself as well, where documents are just treated as raw data, though most of the time it will be appropriate to use subclasses to provide more information and processing operations specific to the datatype.

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for RawDocumentType

keys = ['raw_data']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class TextDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Documents that contain text, most often human-readable documents from a textual corpus. Most often used as a superclass for other, more specific, document types.

This type does not special processing, since the storage format is already a unicode string, which is fine for raw text. However, it serves to indicate that the document represents text (not just any old raw data).

The property *text* provides the text, which is, for this base type, just the raw data. However, subclasses will override this, since their raw data will contain information other than the raw text.

data_point_type_supports_python2 = True

formatters = [('text', 'pimlico.datatypes.corpora.formatters.text.TextDocumentFormatte

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for TextDocumentType

keys = ['text']

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

class RawTextDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.TextDocumentType*

Subclass of TextDocumentType used to indicate that the text hasn't been processed (tokenized, etc). Note that text that has been tokenized, parsed, etc does not use subclasses of this type, so they will not be considered compatible if this type is used as a requirement.

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for RawTextDocumentType

exception DataPointError
Bases: Exception

floats

Corpora consisting of lists of ints. These data point types are useful, for example, for encoding text or other sequence data as integer IDs. They are designed to be fast to read.

class FloatListsDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of float list data: each doc contains lists of float. Unlike `IntegerTableDocumentCorpus`, they are not all constrained to have the same length. The downside is that the storage format (and probably I/O speed) isn't quite as efficient. It's still better than just storing ints as strings or JSON objects.

The floats are stored as C double, which use 8 bytes. At the moment, we don't provide any way to change this. An alternative would be to use C floats, losing precision but (almost) halving storage size.

metadata_defaults = {'bytes': (8, 'Number of bytes to use to represent each int. Defa

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: `pimlico.datatypes.corpora.data_points.Document`

Document class for FloatListsDocumentType

keys = ['lists']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

lists

read_rows(*reader*)

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class FloatListDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of float data: each doc contains a single sequence of floats.

The floats are stored as C doubles, using 8 bytes each.

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for FloatListDocumentType

keys = ['list']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

list

read_rows(*reader*)

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class FloatListsFormatter(*corpus_datatype*)
Bases: *pimlico.cli.browser.tools.formatter.DocumentBrowserFormatter*

DATATYPE
alias of *FloatListsDocumentType*

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class VectorDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Like FloatListDocumentType, but each document has the same number of float values.

Each document contains a single list of floats and each one has the same length. That is, each document is one vector.

The floats are stored as C doubles, using 8 bytes each.

formatters = [('vector', 'pimlico.datatypes.corpora.floats.VectorFormatter')]

metadata_defaults = {'dimensions': (10, 'Number of dimensions in each vector (default

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for VectorDocumentType

keys = ['vector']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class VectorFormatter(*corpus_datatype*)
Bases: *pimlico.cli.browser.tools.formatter.DocumentBrowserFormatter*

DATATYPE = VectorDocumentType()

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

grouped

class GroupedCorpus(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.base.IterableCorpus*

datatype_name = 'grouped_corpus'

document_preprocessors = []

class Reader(**args*, ***kwargs*)
Bases: pimlico.datatypes.corpora.base.Reader

Reader class for GroupedCorpus

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.corpora.base.Setup

Setup class for GroupedCorpus.Reader

data_ready(*base_dir*)
Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

reader_type
alias of *GroupedCorpus.Reader*

get_archive(*archive_name*)
Return a *PimarcReader* for the named archive, or, if using the tar backend, a PimarcTarBackend.

extract_file(*archive_name*, *filename*)
Extract an individual file by archive name and filename.

With the old use of tar to store file, this was not an efficient way of extracting a lot of files. The typical use case of a grouped corpus is to iterate over its files, which is much faster.

Now we're using Pimarc, this is faster. However, jumping a lot between different archives is still slow, as you have to load the index for each archive. A better approach is to load an archive and extract all the files from it you need before loading another.

The reader will cache the most recently used archive, so if you use this method multiple times with the same archive name, it won't reload the index in between.

doc_iter(*start_after=None*, *skip=None*, *name_filter=None*)

archive_iter(*start_after=None*, *skip=None*, *name_filter=None*)
Iterate over corpus archive by archive, yielding for each document the archive name, the document name and the document itself.

Parameters

- **name_filter** – if given, should be a callable that takes two args, an archive name and document name, and returns True if the document should be yielded and False if it should be skipped. This can be preferable to filtering the yielded documents, as it skips all document pre-processing for skipped documents, so speeds up things like random subsampling of a corpus, where the document content never needs to be read in skipped cases
- **start_after** – skip over the first portion of the corpus, until the given document is reached. Should be specified as a pair (archive name, doc name)
- **skip** – skips over the first portion of the corpus, until this number of documents have been seen

list_archive_iter()

list_iter()
Iterate over the list of document names, without processing the doc contents.

In some cases, this could be considerably faster than iterating over all the docs.

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.corpora.base.Writer

Writes a large corpus of documents out to disk, grouping them together in Pimarc archives.

A subtlety is that, as soon as the writer has been initialized, it must be legitimate to initialize a datatype to read the corpus. Naturally, at this point there will be no documents in the corpus, but it allows us to do document processing on the fly by initializing writers and readers to be sure the pre/post-processing is identical to if we were writing the docs to disk and reading them in again.

The reader above allows reading from tar archives for backwards compatibility. However, it is no longer possible to write corpora to tar archives. This has been completely replaced by the new Pimarc archives, which are more efficient to use and allow random access when necessary without huge speed penalties.

metadata_defaults = {'gzip': (False, 'Gzip each document before adding it to the

writer_param_defaults = {'append': (False, 'If True, existing archives and their

add_document(*archive_name*, *doc_name*, *doc*, *metadata=None*)

Add a document to the named archive. All docs should be added to a single archive before moving onto the next. If the archive name is the same as the previous doc added, the doc's data will be appended. Otherwise, the archive is finalized and we move onto the new archive.

Parameters

- **metadata** – dict of metadata values to write with the document. If doc is a document instance, the metadata is taken from there first, but these values will override anything in the doc object's metadata. If doc is a bytes object, the metadata kwarg is used
- **archive_name** – archive name
- **doc_name** – name of document
- **doc** – document instance or bytes object containing document's raw data

flush()

Flush disk write of the archive currently being written.

This used to be called after adding each new file, but slows down the writing massively. Not doing this brings a risk that the written archives are very out of date if a process gets forcibly stopped. However, document map processes are better now than they used to be at recovering from this situation when restarting, so I'm removing this flushing to speed things up.

delete_all_archives()

Check for any already written archives and delete them all to make a fresh start at writing this corpus.

class AlignedGroupedCorpora(*readers*)

Bases: `object`

Iterator for iterating over multiple corpora simultaneously that contain the same files, grouped into archives in the same way. This is the standard utility for taking multiple inputs to a Pimlico module that contain different data but for the same corpus (e.g. output of different tools).

archive_iter(*start_after=None*, *skip=None*, *name_filter=None*)

class GroupedCorpusWithTypeFromInput(*input_name=None*)

Bases: *pimlico.datatypes.base.DynamicOutputDatatype*

Dynamic datatype that produces a GroupedCorpus with a document datatype that is the same as the input's document/data-point type.

If the input name is not given, uses the first input.

Unlike *CorpusWithTypeFromInput*, this does not infer whether the result should be a grouped corpus or not: it always is. The input should be an iterable corpus (or subtype, including grouped corpus), and that's where the datatype will come from.

datatype_name = 'grouped corpus with input doc type'

get_base_datatype()
If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return a datatype instance. By default, it returns None.

If this information is available, it will be used in documentation.

get_datatype(*module_info*)

class CorpusWithTypeFromInput(*input_name=None*)
Bases: *pimlico.datatypes.base.DynamicOutputDatatype*

Infer output corpus' data-point type from the type of an input. Passes the data point type through. Similar to *GroupedCorpusWithTypeFromInput*, but more flexible.

If the input is a grouped corpus, so is the output. Otherwise, it's just an IterableCorpus.

Handles the case where the input is a multiple input. Tries to find a common data point type among the inputs. They must have the same data point type, or all must be subtypes of one of them. (In theory, we could find the most specific common ancestor and use that as the output type, but this is not currently implemented and is probably not worth the trouble.)

Input name may be given. Otherwise, the default input is used.

datatype_name = 'corpus with data-point from input'

get_datatype(*module_info*)

exception CorpusAlignmentError
Bases: Exception

exception GroupedCorpusIterationError
Bases: Exception

ints

Corpora consisting of lists of ints. These data point types are useful, for example, for encoding text or other sequence data as integer IDs. They are designed to be fast to read.

class IntegerListsDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of integer list data: each doc contains lists of ints. Unlike IntegerTableDocumentType, they are not all constrained to have the same length. The downside is that the storage format (and I/O speed) isn't quite as good. It's still better than just storing ints as strings or JSON objects.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

metadata_defaults = {'bytes': (8, 'Number of bytes to use to represent each int. Defa

data_point_type_supports_python2 = True

bytes

signed

row_length_bytes

int_size

length_size

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

struct

length_struct

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: `pimlico.datatypes.corpora.data_points.Document`

Document class for IntegerListsDocumentType

keys = ['lists']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

lists

read_rows(*reader*)

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class IntegerListDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of integer data: each doc contains a single sequence of ints.

Like IntegerListsDocumentType, but each document is treated as a single list of integers.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

metadata_defaults = {'bytes': (8, 'Number of bytes to use to represent each int. Defa

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

struct

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for IntegerListDocumentType

keys = ['list']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

list

read_rows(*reader*)

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class IntegerDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of integer data: each doc contains a single int.

This may be useful, for example, for storing predicted or gold standard class labels for documents.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

metadata_defaults = {'bytes': (8, 'Number of bytes to use to represent each int. Defa

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

struct

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for IntegerDocumentType

keys = ['val']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

list

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

json

class JsonDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Very simple document corpus in which each document is a JSON object.

formatters = [('json', 'pimlico.datatypes.corpora.formatters.json.JsonFormatter')]

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for JsonDocumentType

keys = ['data']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

strings

Documents consisting of strings.

See also:

TextDocumentType and RawTextDocumentType: basic text (i.e. unicode string) document types for normal textual documents.

class LabelDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Simple document type for storing a short label associated with a document.

Identical to TextDocumentType, but distinguished for typechecking, so that only corpora designed to be used as short labels can be used as input where a label corpus is required.

The string label is stored in the label attribute.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for LabelDocumentType

keys = ['label']

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

table

Corpora where each document is a table, i.e. a list of lists, where each row has the same length and each column has a single datatype. This is designed to be fast to read, but is not a very flexible datatype.

get_struct(*bytes*, *signed*, *row_length*)

class IntegerTableDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.RawDocumentType*

Corpus of tabular integer data: each doc contains rows of ints, where each row contains the same number of values. This allows a more compact representation, which doesn't require converting the ints to strings or scanning for line ends, so is quite a bit quicker and results in much smaller file sizes. The downside is that the files are not human-readable.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

metadata_defaults = {'bytes': (8, 'Number of bytes to use to represent each int. Defa

data_point_type_supports_python2 = True

reader_init(*reader*)
Called when a reader is initialized. May be overridden to perform any tasks specific to the data point type that need to be done before the reader starts producing data points.

The super *reader_init()* should be called. This takes care of making reader metadata available in the *metadata* attribute of the data point type instance.

writer_init(*writer*)
Called when a writer is initialized. May be overridden to perform any tasks specific to the data point type that should be done before documents start getting written.

The super *writer_init()* should be called. This takes care of updating the writer's metadata from anything in the instance's *metadata* attribute, for any keys given in the data point type's *metadata_defaults*.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for IntegerTableDocumentType

keys = ['table']

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

table

row_size

read_rows(*reader*)

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

tokenized

class TokenizedDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.data_points.TextDocumentType*

Specialized data point type for documents that have had tokenization applied. It does very little processing - the main reason for its existence is to allow modules to require that a corpus has been tokenized before it's given as input.

Each document is a list of sentences. Each sentence is a list of words.

formatters = [('tokenized_doc', 'pimlico.datatypes.corpora.tokenized.TokenizedDocumentI

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.data_points.Document

Document class for TokenizedDocumentType

keys = ['sentences']

text

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class CharacterTokenizedDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.tokenized.TokenizedDocumentType*

Simple character-level tokenized corpus. The text isn't stored in any special way, but is represented when read internally just as a sequence of characters in each sentence.

If you need a more sophisticated way to handle character-type (or any non-word) units within each sequence, see *SegmentedLinesDocumentType*.

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.tokenized.Document

Document class for CharacterTokenizedDocumentType

sentences

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class SegmentedLinesDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.tokenized.TokenizedDocumentType*

Document consisting of lines, each split into elements, which may be characters, words, or whatever. Rather like a tokenized corpus, but doesn't make the assumption that the elements (words in the case of a tokenized corpus) don't include spaces.

You might use this, for example, if you want to train character-level models on a text corpus, but don't use strictly single-character units, perhaps grouping together certain short character sequences.

Uses the character / to separate elements in the raw data. If a / is found in an element, it is stored as *@slash@*, so this string is assumed not to be used in any element (which seems reasonable enough, generally).

data_point_type_supports_python2 = True

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.tokenized.Document

Document class for SegmentedLinesDocumentType

text

sentences

raw_to_internal(*raw_data*)
Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

word_annotations

Textual corpus type where each word is accompanied by some annotations.

class WordAnnotationsDocumentType(**args*, ***kwargs*)

Bases: *pimlico.datatypes.corpora.tokenized.TokenizedDocumentType*

List of sentences, each consisting of a list of word, each consisting of a tuple of the token and its annotations.

The document type needs to know what fields will be provided, so that it's possible for a module to require a particular set of fields. The field list also tells the reader in which position to find each field.

E.g. the field list "word,lemma,pos" will store values like "walks|walk|VB" for each token. You could also provide "word,pos,lemma" with "walks|VB|walk" and the reader would know where to find the fields it needs.

When a WordAnnotationsDocumentType is used as an input type requirement, it will accept any input corpus that also has a WordAnnotationsDocumentType as its data-point type and includes at least all of the fields specified for the requirement.

So, a requirement of `GroupedCorpus(WordAnnotationsDocumentType(["word", "pos"]))` will match a supplied type of `GroupedCorpus(WordAnnotationsDocumentType(["word", "pos"]))`, or `GroupedCorpus(WordAnnotationsDocumentType(["word", "pos", "lemma]))`, but not `GroupedCorpus(WordAnnotationsDocumentType(["word", "lemma"]))`.

Annotations are given as strings, not other types (like ints). If you want to store e.g. int or float annotations, you need to do the conversion separately, as the encoding and decoding assumes only strings are used.

Annotations may, however, be `None`. This, as well as any linebreaks and tabs in the strings, will be encoded/decoded by the writer/reader.

data_point_type_options = {'fields': {'help': "Names of the annotation fields. These

data_point_type_supports_python2 = True

check_type(*supplied_type*)

Type checking for an iterable corpus calls this to check that the supplied data point type matches the required one (i.e. this instance). By default, the supplied type is simply required to be an instance of the required type (or one of its subclasses).

This may be overridden to introduce other type checks.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)

Bases: `pimlico.datatypes.corpora.tokenized.Document`

Document class for WordAnnotationsDocumentType

keys = ['word_annotations']

text

sentences

get_field(*field*)

Get the given field for every word in every sentence.

Must be one of the fields available in this datatype.

raw_to_internal(*raw_data*)

Take a bytes object containing the raw data for a document, read in from disk, and produce a dictionary containing all the processed data in the document's internal format.

You will often want to call the super method and replace values or add to the dictionary. Whatever you do, make sure that all the internal data that the super type provides is also provided here, so that all of its properties and methods work.

internal_to_raw(*internal_data*)
Take a dictionary containing all the document's data in its internal format and produce a bytes object containing all that data, which can be written out to disk.

class AddAnnotationField(*input_name*, *add_fields*)
Bases: *pimlico.datatypes.base.DynamicOutputDatatype*

Dynamic type constructor that can be used in place of a module's output type. When called (when the output type is needed), dynamically creates a new type that is a corpus with WordAnnotationsDocumentType with the same fields as the named input to the module, with the addition of one or more new ones.

Parameters

- **input_name** – input to the module whose fields we extend
- **add_fields** – field or fields to add, string names

get_datatype(*module_info*)

get_base_datatype()
If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return a datatype instance. By default, it returns None.

If this information is available, it will be used in documentation.

AddAnnotationFields
alias of *pimlico.datatypes.corpora.word_annotations.AddAnnotationField*

class DependencyParsedDocumentType(**args*, ***kwargs*)
Bases: *pimlico.datatypes.corpora.word_annotations.WordAnnotationsDocumentType*

WordAnnotationsDocumentType with fields word, pos, head, deprel for each token.

Convenience wrapper for use as an input requirement where parsed text is needed.

class Document(*data_point_type*, *raw_data=None*, *internal_data=None*, *metadata=None*)
Bases: pimlico.datatypes.corpora.word_annotations.Document

Document class for DependencyParsedDocumentType

Module contents

Submodules

arrays

Wrappers around Numpy arrays and Scipy sparse matrices.

class NumpyArray(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

datatype_name = 'numpy_array'

datatype_supports_python2 = True

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for NumpyArray

array

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for NumpyArray.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *NumpyArray.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for NumpyArray

write_array(*array*)

metadata_defaults = {}

writer_param_defaults = {}

class ScipySparseMatrix(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

Wrapper around Scipy sparse matrices. The matrix loaded is always in COO format – you probably want to convert to something else before using it. See scipy docs on sparse matrix conversions.

datatype_name = 'scipy_sparse_array'

datatype_supports_python2 = True

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for ScipySparseMatrix

array

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for ScipySparseMatrix.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *ScipySparseMatrix.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for ScipySparseMatrix

write_matrix(*mat*)

metadata_defaults = {}

writer_param_defaults = {}

base

Datatypes provide interfaces for reading and writing datasets. They provide different ways of reading in or iterating over datasets and different ways to write out datasets, as appropriate to the datatype. They are used by Pimlico to typecheck connections between modules to make sure that the output from one module provides a suitable type of data for the input to another. They are then also used by the modules to read in their input data coming from earlier in a pipeline and to write out their output data, to be passed to later modules.

See *Datatypes* for a guide to how Pimlico datatypes work.

This module defines the base classes for all datatypes.

class PimlicoDatatype(**args*, ***kwargs*)
Bases: object

The abstract superclass of all datatypes. Provides basic functionality for identifying where data should be stored and such.

Datatypes are used to specify the routines for reading the output from modules, via their reader class.

module is the ModuleInfo instance for the pipeline module that this datatype was produced by. It may be None, if the datatype wasn't instantiated by a module. It is not required to be set if you're instantiating a datatype in some context other than module output. It should generally be set for input datatypes, though, since they are treated as being created by a special input module.

If you're **creating a new datatype**, refer to the *datatype documentation*.

datatype_options = {}
Options specified in the same way as module options that control the nature of the datatype. These are not things to do with reading of specific datasets, for which the dataset's metadata should be used. These are things that have an impact on typechecking, such that options on the two checked datatypes are required to match for the datatypes to be considered compatible.

They should always be an ordered dict, so that they can be specified using positional arguments as well as kwargs and config parameters.

shell_commands = []
Override to provide shell commands specific to this datatype. Should include the superclass' list.

datatype_supports_python2 = True
Most core Pimlico datatypes support use in Python 2 and 3. Datatypes that do should set this to True. If it is False, the datatype is assumed to work only in Python 3.

Python 2 compatibility requires extra work from the programmer. Datatypes should generally declare whether or not they provide this support by overriding this explicitly.

Use `supports_python2()` to check whether a datatype instance supports Python 2. (There may be reasons for a datatype's instance to override this class-level setting.)

datatype_name = 'base_datatype'
Identifier (without spaces) to distinguish this datatype

supports_python2()
By default, just returns cls.datatype_supports_python2. Subclasses might override this.

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

get_writer_software_dependencies()
Get a list of all software required to **write** this datatype using its *Writer*. This works in a similar way to *get_software_dependencies()* (for the *Reader*) and the dependencies will be check before the writer is instantiated.

It is assumed that all the reader's dependencies also apply to the writer, so this method only needs to specify any additional dependencies the writer has.

You should call the super method for checking superclass dependencies.

get_writer(*base_dir*, *pipeline*, *module=None*, ***kwargs*)
Instantiate a writer to write data to the given base dir.

Kwargs are passed through to the writer and used to specify initial metadata and writer params.

Parameters

- **base_dir** – output dir to write dataset to
- **pipeline** – current pipeline
- **module** – module name (optional, for debugging only)

Returns instance of the writer subclass corresponding to this datatype

classmethod instantiate_from_options(*options={}*)
Given string options e.g. from a config file, perform option processing and instantiate datatype

classmethod datatype_full_class_name()
The fully qualified name of the class for this datatype, by which it is reference in config files. Generally, datatypes don't need to override this, but type requirements that take the place of datatypes for type checking need to provide it.

check_type(*supplied_type*)
Method used by datatype type-checking algorithm to determine whether a supplied datatype (given as an instance of a subclass of PimlicoDatatype) is compatible with the present datatype, which is being treated as a type requirement.

Typically, the present class is a type requirement on a module input and *supplied_type* is the type provided by a previous module's output.

The default implementation simply checks whether *supplied_type* is a subclass of the present class. Subclasses may wish to impose different or additional checks.

Parameters **supplied_type** – type provided where the present class is required, or datatype instance

Returns True if the check is successful, False otherwise

type_checking_name()
Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

full_datatype_name()
Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

run_browser(*reader*, *opts*)
Launches a browser interface for reading this datatype, browsing the data provided by the given reader.

Not all datatypes provide a browser. For those that don't, this method should raise a NotImplementedError.

opts provides the argparser options from the command line.

This tool used to be only available for iterable corpora, but now it's possible for any datatype to provide a browser. IterableCorpus provides its own browser, as before, which uses one of the data point type's formatters to format documents.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: `object`

The abstract superclass of all dataset readers.

You do not need to subclass or instantiate these yourself: subclasses are created automatically to correspond to each datatype. You can add functionality to a datatype's reader by creating a nested *Reader* class. This will inherit from the parent datatype's reader. This happens automatically - you don't need to do it yourself and shouldn't inherit from anything:

```
class MyDatatype(PimlicoDatatype):
    class Reader:
        # Override reader things here
```

process_setup()
Do any processing of the setup object (e.g. retrieving values and setting attributes on the reader) that should be done when the reader is instantiated.

get_detailed_status()
Returns a list of strings, containing detailed information about the data.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

class Setup(*datatype*, *data_paths*)
Bases: `object`

Abstract superclass of all dataset reader setup classes.

See *Datatypes* for a information about how this class is used.

These classes provide any functionality relating to a reader needed before it is ready to read and instantiated. Most importantly, it provides the *ready_to_read()* method, which indicates whether the reader is ready to be instantiated.

The standard implementation, which can be used in almost all cases, takes a list of possible paths to the dataset at initialization and checks whether the dataset is ready to be read from any of them. You generally don't need to override *ready_to_read()* with this, but just *data_ready()*, which checks whether the data is ready to be read in a specific location. You can call the parent class' data-ready checks using super: *super(MyDatatype.Reader.Setup, self).data_ready()*.

The whole *Setup* object will be passed to the corresponding *Reader*'s init, so that it has access to data locations, etc.

Subclasses may take different init args/kwargs and store whatever attributes are relevant for preparing their corresponding *Reader*. In such cases, you will usually override a *ModuleInfo*'s *get_output_reader_setup()* method for a specific output's reader preparation, to provide it with the appropriate arguments. Do this by calling the *Reader* class' *get_setup(*args, **kwargs)* class method, which passes args and kwargs through to the *Setup*'s init.

You do not need to subclass or instantiate these yourself: subclasses are created automatically to correspond to each reader type. You can add functionality to a reader's setup by creating a nested *Setup* class. This will inherit from the parent reader's setup. This happens automatically - you don't need to do it yourself and shouldn't inherit from anything:

```
class MyDatatype(PimlicoDatatype):
    class Reader:
        # Overide reader things here

        class Setup:
            # Override setup things here
            # E.g.:
            def data_ready(path):
                # Parent checks: usually you want to do this
                if not super(MyDatatype.Reader.Setup, self).data_
↪ready(path):
                    return False
                # Check whether the data's ready according to our own␣
↪criteria
```

(continues on next page)

(continued from previous page)

```
        # ...
        return True
```

The first arg to the init should always be the datatype instance.

reader_type
alias of *PimlicoDatatype.Reader*

data_ready(*path*)
Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.

Returns True if the reader's ready to be instantiated, False otherwise

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

get_base_dir()

Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()

Returns the path to the data dir within the base dir (typically a dir called "data")

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

get_reader(*pipeline*, *module=None*)
Instantiate a reader using this setup.

Parameters

- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

classmethod get_setup(*datatype*, **args*, ***kwargs*)
Instantiate a reader setup object for this reader. The args and kwargs are those of the reader's corresponding setup class and will be passed straight through to the init.

metadata
Read in metadata from a file in the corpus directory.

Note that this is no longer cached in memory. We need to be sure that the metadata values returned are always up to date with what is on disk, so always re-read the file when we need to get a value

from the metadata. Since the file is typically small, this is unlikely to cause a problem. If we decide to return to cacheing the metadata dictionary in future, we will need to make sure that we can never run into problems with out-of-date metadata being returned.

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)

Bases: `object`

The abstract superclass of all dataset writers.

You do not need to subclass or instantiate these yourself: subclasses are created automatically to correspond to each datatype. You can add functionality to a datatype's writer by creating a nested *Writer* class. This will inherit from the parent datatype's writer. This happens automatically - you don't need to do it yourself and shouldn't inherit from anything:

```
class MyDatatype(PimlicoDatatype):
    class Writer:
        # Overide writer things here
```

Writers should be used as context managers. Typically, you will get hold of a writer for a module's output directly from the module-info instance:

```
with module.get_output_writer("output_name") as writer:
    # Call the writer's methods, set its attributes, etc
    writer.do_something(my_data)
    writer.some_attr = "This data"
```

Any additional kwargs passed into the writer (which you can do by passing kwargs to `get_output_writer()` on the module) will set values in the dataset's metadata. Available parameters are given, along with their default values, in the dictionary `metadata_defaults` on a Writer class. They also include all values from ancestor writers.

It is important to pass in parameters as kwargs that affect the writing of the data, to ensure that the correct values are available as soon as the writing process starts.

All metadata values, including those passed in as kwargs, should be serializable as simple JSON types.

Another set of parameters, *writer params*, is used to specify things that affect the writing process, but do not need to be stored in the metadata. This could be, for example, the number of CPUs to use for some part of the writing process. Unlike, for example, the format of the stored data, this is not needed later when the data is read.

Available writer params are given, along with their default values, in the dictionary `writer_param_defaults` on a Writer class. (They do not need to be JSON serializable.) Their values are also specified as kwargs in the same way as metadata.

metadata_defaults = {}

writer_param_defaults = {}

required_tasks = []

This can be overriden on writer classes to add this list of tasks to the required tasks when the writer is initialized

require_tasks(**tasks*)

Add a name or multiple names to the list of output tasks that must be completed before writing is finished

task_complete(*task*)

Mark the named task as completed

incomplete_tasks
List of required tasks that have not yet been completed

write_metadata()

class DynamicOutputDatatype
Bases: object

Types of module outputs may be specified as an instance of a subclass of *PimlicoDatatype*, or alternatively as an instance of DynamicOutputType. In this case, get_datatype() is called when the output datatype is needed, passing in the module info instance for the module, so that a specialized datatype can be produced on the basis of options, input types, etc.

The dynamic type must provide certain pieces of information needed for typechecking.

If a base datatype is available (i.e. indication of the datatype before the module is instantiated), we take the information regarding whether the datatype supports Python 2 from there. If not, we assume it does. This may seems the opposite to other places: for example, the base datatype says it does **not** support Python 2 and subclasses must declare if they do. However, dynamic output datatypes are often used with modules that work with a broad range of input datatypes. It is therefore wrong to say that they do not support Python 2, since they will provided the input module does.

datatype_name = None

get_datatype(*module_info*)

get_base_datatype()
If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return a datatype instance. By default, it returns None.

If this information is available, it will be used in documentation.

supports_python2()

class DynamicInputDatatypeRequirement
Bases: object

Types of module inputs may be given as an instance of a subclass of *PimlicoDatatype*, a tuple of datatypes, or an instance a DynamicInputDatatypeRequirement subclass. In this case, check_type(supplied_type) is called during typechecking to check whether the type that we've got conforms to the input type requirements.

Additionally, if datatype_doc_info is provided, it is used to represent the input type constraints in documentation.

datatype_doc_info = None

check_type(*supplied_type*)

type_checking_name()
Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Subclasses may want to override this too.

class MultipleInputs(*datatype_requirements*)
Bases: object

A wrapper around an input datatype that can be used as an item in a module's inputs, which lets the module accept an unbounded number of inputs, all satisfying the same datatype requirements.

When writing the inputs in a config file, they can be specified as a comma-separated list of the usual type of specification (module name, with optional output name). Each item in the list must point to a dataset (module output) that satisfies the type-checking for the wrapped datatype.

```
[module3]
type=pimlico.modules.some_module
input_datasets=module1.the_output,module2.the_output
```

Here `module1`'s output `the_output` and `module2`'s output `the_output` must both be of valid types for the multiple-input `datasets` to this module.

The list may also include (or entirely consist of) a base module name from the pipeline that has been **expanded** into multiple modules according to **alternative parameters** (the type separated by vertical bars, see *Multiple parameter values*). You can use the notation `*name`, where `name` is the base module name, to denote all of the expanded module names as inputs. These are treated as if you'd written out all of the expanded module names separated by commas.

```
[module1]
type=pimlico.modules.any_module
param={case1}first value for param|{case2}second value

[module3]
type=pimlico.modules.some_module
input_datasets=*module1.the_output
```

Here `module1` will be expanded into `module1[case1]` and `module1[case2]`, each having a different value for option `param`. The `*`-notation is a shorthand to say that the input `datasets` should get the output `the_output` from **both** of these alternatives, as if you had written `module1[case1].the_output, module1[case2].the_output`.

If a module provides multiple outputs, all of a suitable type, that you want to feed into the same (multiple-input) input, you can specify a list of **all of the module's outputs** using the notation `module_name.*`.

```
# This module provides two outputs, output1 and output2
[module2]
type=pimlico.modules.multi_output_module

[module3]
type=pimlico.modules.some_module
input_datasets=module2.*
```

is equivalent to:

```
[module3]
type=pimlico.modules.some_module
input_datasets=module2.output1,module2.output2
```

If you need the **same input specification to be repeated** multiple times in a list, instead of writing it out explicitly you can use a multiplier to repeat it N times by putting `*N` after it. This is particularly useful when `N` is the result of expanding module variables, allowing the number of times an input is repeated to depend on some modvar expression.

```
[module3]
type=pimlico.modules.some_module
input_datasets=module1.the_output*3
```

is equivalent to:

```
[module3]
type=pimlico.modules.some_module
input_datasets=module1.the_output,module1.the_output,module1.the_output
```

When *get_input()* is called on the module info, if multiple inputs have been provided, instead of returning a single dataset reader, a list of readers is returned. You can use `get_input(input_name, always_list=True)` to always return a list of readers, even if only a single dataset was given as input. This is usually the best way to handle multiple inputs in module code.

supports_python2()

exception DatatypeLoadError
Bases: `Exception`

exception DatatypeWriteError
Bases: `Exception`

core

Some basic core datatypes that are commonly used for passing simple data, like strings and dicts, through pipelines.

class Dict(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Simply stores a Python dict, pickled to disk. All content in the dict should be pickleable.

datatype_name = 'dict'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: `pimlico.datatypes.base.Reader`

Reader class for Dict

class Setup(*datatype*, *data_paths*)
Bases: `pimlico.datatypes.base.Setup`

Setup class for Dict.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *Dict.Reader*

get_dict()

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: `pimlico.datatypes.base.Writer`

Writer class for Dict

required_tasks = ['dict']

write_dict(*d*)

metadata_defaults = {}

writer_param_defaults = {}

class StringList(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Simply stores a Python list of strings, written out to disk in a readable form. Not the most efficient format, but if the list isn't humungous it's OK (e.g. storing vocabularies).

datatype_name = 'string_list'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for StringList

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for StringList.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *StringList.Reader*

get_list()

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for StringList

required_tasks = ['list']

write_list(*l*)

metadata_defaults = {}

writer_param_defaults = {}

dictionary

This module implements the concept of a Dictionary – a mapping between words and their integer ids.

The implementation is based on Gensim, because Gensim is wonderful and there's no need to reinvent the wheel. We don't use Gensim's data structure directly, because it's unnecessary to depend on the whole of Gensim just for one data structure.

However, it is possible to retrieve a Gensim dictionary directly from the Pimlico data structure if you need to use it with Gensim.

class Dictionary(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Dictionary encapsulates the mapping between normalized words and their integer ids. This class is responsible for reading and writing dictionaries.

DictionaryData is the data structure itself, which is very closely related to Gensim's dictionary.

datatype_name = 'dictionary'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for Dictionary

get_data()
Load the dictionary and return a *DictionaryData* object.

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for Dictionary.Reader

get_required_paths()
Require the dictionary file to be written

reader_type
alias of *Dictionary.Reader*

get_detailed_status()
Returns a list of strings, containing detailed information about the data.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

When the context manager is created, a new, empty *DictionaryData* instance is created. You can build your dictionary by calling *add_documents()* on the writer, or accessing the dictionary data structure directly (via the *data* attribute), or simply replace it with a fully formed *DictionaryData* instance of your own, using the same instance.

You can specify a list/set of stopwords when instantiating the writer. These will be excluded from the dictionary if seen in the corpus.

add_documents(*documents*, *prune_at=2000000*)

filter(*threshold=None*, *no_above=None*, *limit=None*)

filter_high_low(*threshold=None*, *no_above=None*, *limit=None*)

metadata_defaults = {}

writer_param_defaults = {}

run_browser(*reader*, *opts*)
Browse the vocab simply by printing out all the words

class DictionaryData
Bases: object

Dictionary encapsulates the mapping between normalized words and their integer ids. This is taken almost directly from Gensim.

We also store a set of stopwords. These can be set explicitly (see *add_stopwords()*), and will also include any words that are removed as a result of filters on the basis that they're too common. This means that we can tell which words are OOV because we've never seen them (or not seen them often) and which are common but filtered.

id2token

keys()
Return a list of all token ids.

refresh_id2token()

add_stopwords(*new_stopwords*)
Add some stopwords to the list.

Raises an error if a stopword is in the dictionary. We don't remove the term here, because that would end up changing IDs of other words unexpectedly. Instead, we leave it to the user to ensure a stopword is removed before being added to the list.

Terms already in the stopword list will not be added to the dictionary later.

add_term(*term*)
Add a term to the dictionary, without any occurrence count. Note that if you run threshold-based filters after adding a term like this, it will get removed.

add_documents(*documents*, *prune_at=2000000*)
Update dictionary from a collection of documents. Each document is a list of tokens = **tokenized and normalized** strings (either utf8 or unicode).

This is a convenience wrapper for calling *doc2bow* on each document with *allow_update=True*, which also prunes infrequent words, keeping the total number of unique words <= *prune_at*. This is to save memory on very large inputs. To disable this pruning, set *prune_at=None*.

Keeps track of total documents added, rather than just those added in this call, to decide when to prune. Otherwise, making many calls with a small number of docs in each results in pruning on every call.

doc2bow(*document*, *allow_update=False*, *return_missing=False*)
Convert *document* (a list of words) into the bag-of-words format = list of *(token_id, token_count)* 2-tuples. Each word is assumed to be a **tokenized and normalized** string (either unicode or utf8-encoded). No further preprocessing is done on the words in *document*; apply tokenization, stemming etc. before calling this method.

If *allow_update* is set, then also update dictionary in the process: create ids for new words. At the same time, update document frequencies – for each word appearing in this document, increase its document frequency (*self.dfs*) by one.

If *allow_update* is **not** set, this function is *const*, aka read-only.

filter_extremes(*no_below=5*, *no_above=0.5*, *keep_n=100000*)
Filter out tokens that appear in

1. fewer than *no_below* documents (absolute number) or
2. more than *no_above* documents (fraction of total corpus size, *not* absolute number).
3. after (1) and (2), keep only the first *keep_n* most frequent tokens (or keep all if *None*).

After the pruning, shrink resulting gaps in word ids.

Note: Due to the gap shrinking, the same word may have a different word id before and after the call to this function!

filter_high_low_extremes(*no_below=5*, *no_above=0.5*, *keep_n=100000*, *add_stopwords=True*)
Filter out tokens that appear in

1. fewer than *no_below* documents (absolute number) or
2. more than *no_above* documents (fraction of total corpus size, *not* absolute number).
3. after (1) and (2), keep only the first *keep_n* most frequent tokens (or keep all if *None*).

This is the same as filter_extremes(), but returns a separate list of terms removed because they're too frequent and those removed because they're not frequent enough.

If *add_stopwords=True* (default), any frequent words filtered out will be added to the stopwords list.

filter_tokens(*bad_ids=None*, *good_ids=None*)
Remove the selected *bad_ids* tokens from all dictionary mappings, or, keep selected *good_ids* in the mapping and remove the rest.

bad_ids and *good_ids* are collections of word ids to be removed.

compactify()
Assign new word ids to all words.

This is done to make the ids more compact, e.g. after some tokens have been removed via *filter_tokens()* and there are gaps in the id series. Calling this method will remove the gaps.

as_gensim_dictionary()
Convert to Gensim's dictionary type, which this type is based on. If you call this, Gensim will be imported, so your code becomes dependent on having Gensim installed.

Returns gensim dictionary

embeddings

Datatypes to store embedding vectors, together with their words.

The main datatype here, *Embeddings*, is the main datatype that should be used for passing embeddings between modules.

We also provide a simple file collection datatype that stores the files used by Tensorflow, for example, as input to the Tensorflow Projector. Modules that need data in this format can use this datatype, which makes it easy to convert from other formats.

class Vocab(*word*, *index*, *count=0*)
Bases: object

A single vocabulary item, used internally for collecting per-word frequency info. A simplified version of Gensim's Vocab.

class Embeddings(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Datatype to store embedding vectors, together with their words. Based on Gensim's KeyedVectors object, but adapted for use in Pimlico and so as not to depend on Gensim. (This means that this can be used more generally for storing embeddings, even when we're not depending on Gensim.)

Provides a method to map to Gensim's KeyedVectors type for compatibility.

Doesn't provide all of the functionality of KeyedVectors, since the main purpose of this is for storage of vectors and other functionality, like similarity computations, can be provided by utilities or by direct use of Gensim.

Since we don't depend on Gensim, this datatype supports Python 2. However, if you try to use the mapping to Gensim's type, this will only work with Gensim installed and therefore also depends on Python 3.

datatype_name = 'embeddings'

datatype_supports_python2 = True

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

get_writer_software_dependencies()
Get a list of all software required to **write** this datatype using its *Writer*. This works in a similar way to *get_software_dependencies()* (for the *Reader*) and the dependencies will be check before the writer is instantiated.

It is assumed that all the reader's dependencies also apply to the writer, so this method only needs to specify any additional dependencies the writer has.

You should call the super method for checking superclass dependencies.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: `pimlico.datatypes.base.Reader`

Reader class for Embeddings

class Setup(*datatype*, *data_paths*)
Bases: `pimlico.datatypes.base.Setup`

Setup class for Embeddings.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *Embeddings.Reader*

vectors

normed_vectors

vector_size

word_counts

index2vocab

index2word

vocab

word_vec(*word*, *norm=False*)
Accept a single word as input. Returns the word's representation in vector space, as a 1D numpy array.

word_vecs(*words*, *norm=False*)
Accept multiple words as input. Returns the words' representations in vector space, as a 1D numpy array.

to_keyed_vectors()

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: `pimlico.datatypes.base.Writer`

Writer class for Embeddings

required_tasks = ['vocab', 'vectors']

write_vectors(*arr*)
Write out vectors from a Numpy array

write_word_counts(*word_counts*)
Write out vocab from a list of words with counts.

Parameters **word_counts** – list of (unicode, int) pairs giving each word and its count. Vocab indices are determined by the order of words

write_vocab_list(*vocab_items*)
Write out vocab from a list of vocab items (see `Vocab`).

Parameters **vocab_items** – list of `Vocab` s

write_keyed_vectors(**kvecs*)
Write both vectors and vocabulary straight from Gensim's `KeyedVectors` data structure. Can accept multiple objects, which will then be concatenated in the output.

metadata_defaults = {}

writer_param_defaults = {}

run_browser(*reader*, *opts*)
Just output some info about the embeddings.

We could also iterate through some of the words or provide other inspection tools, but for now we don't do that.

class TSVVecFiles(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

Embeddings stored in TSV files. This format is used by Tensorflow and can be used, for example, as input to the Tensorflow Projector.

It's just a TSV file with each vector on a row, and another metadata TSV file with the names associated with the points and the counts. The counts are not necessary, so the metadata can be written without them if necessary.

datatype_name = 'tsv_vec_files'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: `pimlico.datatypes.files.Reader`

Reader class for TSVVecFiles

get_embeddings_data()

get_embeddings_metadata()

class Setup(*datatype*, *data_paths*)
Bases: `pimlico.datatypes.files.Setup`

Setup class for TSVVecFiles.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *TSVVecFiles.Reader*

class Writer(**args*, ***kwargs*)
Bases: `pimlico.datatypes.files.Writer`

Writer class for TSVVecFiles

write_vectors(*array*)

write_vocab_with_counts(*word_counts*)

write_vocab_without_counts(*words*)

metadata_defaults = {}

writer_param_defaults = {}

class Word2VecFiles(**args*, ***kwargs*)

Bases: *pimlico.datatypes.files.NamedFileCollection*

datatype_name = 'word2vec_files'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)

Bases: pimlico.datatypes.base.Reader

Reader class for NamedFileCollection

class Setup(*datatype*, *data_paths*)

Bases: pimlico.datatypes.base.Setup

Setup class for NamedFileCollection.Reader

get_required_paths()

May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type

alias of NamedFileCollection.Reader

absolute_filenames

For backwards compatibility: use absolute_paths by preference

absolute_paths

get_absolute_path(*filename*)

open_file(*filename=None*, *mode='r'*)

process_setup()

Do any processing of the setup object (e.g. retrieving values and setting attributes on the reader) that should be done when the reader is instantiated.

read_file(*filename=None*, *mode='r'*, *text=False*)

Read a file from the collection.

Parameters

- **filename** – string filename, which should be one of the filenames specified for this collection; or an integer, in which case the ith file in the collection is read. If not given, the first file is read
- **mode** –
- **text** – if True, the file is treated as utf-8-encoded text and a unicode object is returned. Otherwise, a bytes object is returned.

Returns

read_files(*mode='r'*, *text=False*)

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for NamedFileCollection

absolute_paths

file_written(*filename*)
Mark the given file as having been written, if write_file() was not used to write it.

get_absolute_path(*filename=None*)

metadata_defaults = {}

open_file(*filename=None*)

write_file(*filename*, *data*, *text=False*)
If text=True, the data is expected to be unicode and is encoded as utf-8. Otherwise, data should be a bytes object.

writer_param_defaults = {}

class DocEmbeddingsMapper(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Abstract datatype.

An embedding loader provides a method to take a list of tokens (e.g. a tokenized document) and produce an embedding for each token. It will not necessarily be able to produce an embedding for *any* given term, so might return None for some tokens.

This is more general than the *Embeddings* datatype, as it allows this method to potentially produce embeddings for an infinite set of terms. Conversely, it is not able to say which set of terms it can produce embeddings for.

It provides a unified interface to composed embeddings, like fastText, which can use sub-word information to produce embeddings of OOVs; context-sensitive embeddings, like BERT, which taken into account the context of a token; and fixed embeddings, which just return a fixed embedding for in-vocab terms.

Some subtypes are just wrappers for fixed sets of embeddings.

datatype_name = 'doc_embeddings_mapper'

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

run_browser(*reader*, *opts*)
Simple tool to display embeddings for the words of user-entered sentences.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)

Bases: pimlico.datatypes.base.Reader

Reader class for DocEmbeddingsMapper

get_embeddings(*tokens*)

Subclasses should produce a list, with an item for each token. The item may be None, or a numpy array containing a vector for the token.

Parameters **tokens** – list of strings

Returns list of embeddings

class Setup(*datatype*, *data_paths*)

Bases: pimlico.datatypes.base.Setup

Setup class for DocEmbeddingsMapper.Reader

data_ready(*path*)

Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()

Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()

Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)

Instantiate a reader using this setup.

Parameters

- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()

May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)

Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type

alias of *DocEmbeddingsMapper.Reader*

ready_to_read()

Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.

Returns True if the reader's ready to be instantiated, False otherwise

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)

Bases: `object`

The abstract superclass of all dataset writers.

You do not need to subclass or instantiate these yourself: subclasses are created automatically to correspond to each datatype. You can add functionality to a datatype's writer by creating a nested *Writer* class. This will inherit from the parent datatype's writer. This happens automatically - you don't need to do it yourself and shouldn't inherit from anything:

```
class MyDatatype(PimlicoDatatype):
    class Writer:
        # Overide writer things here
```

Writers should be used as context managers. Typically, you will get hold of a writer for a module's output directly from the module-info instance:

```
with module.get_output_writer("output_name") as writer:
    # Call the writer's methods, set its attributes, etc
    writer.do_something(my_data)
    writer.some_attr = "This data"
```

Any additional kwargs passed into the writer (which you can do by passing kwargs to `get_output_writer()` on the module) will set values in the dataset's metadata. Available parameters are given, along with their default values, in the dictionary `metadata_defaults` on a Writer class. They also include all values from ancestor writers.

It is important to pass in parameters as kwargs that affect the writing of the data, to ensure that the correct values are available as soon as the writing process starts.

All metadata values, including those passed in as kwargs, should be serializable as simple JSON types.

Another set of parameters, *writer params*, is used to specify things that affect the writing process, but do not need to be stored in the metadata. This could be, for example, the number of CPUs to use for some part of the writing process. Unlike, for example, the format of the stored data, this is not needed later when the data is read.

Available writer params are given, along with their default values, in the dictionary `writer_param_defaults` on a Writer class. (They do not need to be JSON serializable.) Their values are also specified as kwargs in the same way as metadata.

incomplete_tasks
List of required tasks that have not yet been completed

metadata_defaults = {}

require_tasks(**tasks*)
Add a name or multiple names to the list of output tasks that must be completed before writing is finished

required_tasks = []

task_complete(*task*)
Mark the named task as completed

write_metadata()

writer_param_defaults = {}

class FastTextDocMapper(**args*, ***kwargs*)

Bases: *pimlico.datatypes.embeddings.DocEmbeddingsMapper*

datatype_name = 'fasttext_doc_embeddings_mapper'

get_software_dependencies()

Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)

Bases: pimlico.datatypes.embeddings.Reader

Reader class for FastTextDocMapper

model

get_embeddings(*tokens*)

Subclasses should produce a list, with an item for each token. The item may be None, or a numpy array containing a vector for the token.

Parameters **tokens** – list of strings

Returns list of embeddings

class Setup(*datatype*, *data_paths*)

Bases: pimlico.datatypes.embeddings.Setup

Setup class for FastTextDocMapper.Reader

data_ready(*path*)

Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()

Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()

Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)
Instantiate a reader using this setup.

Parameters
- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type
alias of *FastTextDocMapper.Reader*

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.

Returns True if the reader's ready to be instantiated, False otherwise

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for FastTextDocMapper

required_tasks = ['model']

save_model(*model*)

metadata_defaults = {}

writer_param_defaults = {}

class FixedEmbeddingsDocMapper(**args*, ***kwargs*)
Bases: *pimlico.datatypes.embeddings.DocEmbeddingsMapper*

datatype_name = 'fixed_embeddings_doc_embeddings_mapper'

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.embeddings.Reader

Reader class for FixedEmbeddingsDocMapper

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.embeddings.Setup

Setup class for FixedEmbeddingsDocMapper.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *FixedEmbeddingsDocMapper.Reader*

vectors

vector_size

word_counts

index2vocab

index2word

vocab

word_vec(*word*)
Accept a single word as input. Returns the word's representation in vector space, as a 1D numpy array.

get_embeddings(*tokens*)
Subclasses should produce a list, with an item for each token. The item may be None, or a numpy array containing a vector for the token.

Parameters **tokens** – list of strings

Returns list of embeddings

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: `pimlico.datatypes.base.Writer`

Writer class for FixedEmbeddingsDocMapper

required_tasks = ['vocab', 'vectors']

write_vectors(*arr*)
Write out vectors from a Numpy array

write_word_counts(*word_counts*)
Write out vocab from a list of words with counts.

Parameters **word_counts** – list of (unicode, int) pairs giving each word and its count. Vocab indices are determined by the order of words

write_vocab_list(*vocab_items*)
Write out vocab from a list of vocab items (see `Vocab`).

Parameters **vocab_items** – list of `Vocab` s

write_keyed_vectors(**kvecs*)
Write both vectors and vocabulary straight from Gensim's `KeyedVectors` data structure. Can accept multiple objects, which will then be concatenated in the output.

metadata_defaults = {}

writer_param_defaults = {}

features

class ScoredRealFeatureSets(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

Sets of features, where each feature has an associated real number value, and each set (i.e. data point) has a score.

This is suitable as training data for a multidimensional regression.

Stores a dictionary of feature types and uses integer IDs to refer to them in the data storage.

Todo: Add unit test for ScoredReadFeatureSets

---

datatype_name = 'scored_real_feature_sets'

datatype_supports_python2 = True

browse_file(*reader*, *filename*)
Return text for a particular file in the collection to show in the browser. By default, just reads in the file's data and returns it, but subclasses might want to override this (perhaps conditioned on the filename) to format the data readably.

Parameters

- **reader** –
- **filename** –

Returns file data to show

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for ScoredRealFeatureSets

read_samples()
Read all samples in from the data file.

Note that *__iter__()* iterates over the file without loading everything into memory, which may be preferable if dealing with big datasets.

iter_ids()
Iterate over the raw ID data from the data file, without translating feature type IDs into feature names.

feature_types

num_samples

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for ScoredRealFeatureSets.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *ScoredRealFeatureSets.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for ScoredRealFeatureSets

set_feature_types(*feature_types*)
Explicitly set the list of feature types that will be written out. All feature types given will be included, plus possibly others that are used in the written samples, which will be added to the set.

This can be useful if you want your feature vocabulary to include the whole of a given set, even if some feature types are never used in the data. It can also be useful to ensure particular IDs are used for particular feature types, if you care about that.

write_samples(*samples*)
Writes a list of samples, each given as a (features, score) pair. See *write_sample()*

write_sample(*features*, *score*)
Write out a single sample to the end of the data file. Features should be given by name in a dictionary mapping the feature type to its value.

Parameters

- **features** – dict(feature name -> feature value)
- **score** – score associated with this data point

metadata_defaults = {}

writer_param_defaults = {}

files

File collections and files.

There used to be an UnnamedFileCollection, which has been removed in the move to the new datatype system. It used to be used mostly for input datatypes, which don't exist any more. There may still be a use for this, though, so I may be added in future.

class NamedFileCollection(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Datatypes that stores a fixed collection of files, which have fixed names (or at least names that can be determined from the class). Very many datatypes fall into this category. Overriding this base class provides them with some common functionality, including the possibility of creating a union of multiple datatypes.

The datatype option `filenames` should specify a list of filenames contained by the datatype. For typechecking, the provided type must have at least all the filenames of the type requirement, though it may include more.

All files are contained in the datatypes data directory. If files are stored in subdirectories, this may be specified in the list of filenames using `/` s. (Always use forward slashes, regardless of the operating system.)

datatype_name = 'named_file_collection'

datatype_options = {'filenames': {'default': [], 'help': 'Filenames contained in th

datatype_supports_python2 = True

check_type(*supplied_type*)
Method used by datatype type-checking algorithm to determine whether a supplied datatype (given as an instance of a subclass of PimlicoDatatype) is compatible with the present datatype, which is being treated as a type requirement.

Typically, the present class is a type requirement on a module input and *supplied_type* is the type provided by a previous module's output.

The default implementation simply checks whether *supplied_type* is a subclass of the present class. Subclasses may wish to impose different or additional checks.

Parameters **supplied_type** – type provided where the present class is required, or datatype instance

Returns True if the check is successful, False otherwise

browse_file(*reader*, *filename*)
Return text for a particular file in the collection to show in the browser. By default, just reads in the file's data and returns it, but subclasses might want to override this (perhaps conditioned on the filename) to format the data readably.

Parameters

- **reader** –
- **filename** –

Returns file data to show

run_browser(*reader*, *opts*)
All NamedFileCollections provide a browser that just lets you see a list of the files and view them, in the case of text files.

Subclasses may override the way individual files are shown by overriding *browse_file()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for NamedFileCollection

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for NamedFileCollection.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *NamedFileCollection.Reader*

process_setup()
Do any processing of the setup object (e.g. retrieving values and setting attributes on the reader) that should be done when the reader is instantiated.

get_absolute_path(*filename*)

absolute_paths

absolute_filenames
For backwards compatibility: use absolute_paths by preference

read_file(*filename=None*, *mode='r'*, *text=False*)
Read a file from the collection.

Parameters

- **filename** – string filename, which should be one of the filenames specified for this collection; or an integer, in which case the ith file in the collection is read. If not given, the first file is read
- **mode** –
- **text** – if True, the file is treated as utf-8-encoded text and a unicode object is returned. Otherwise, a bytes object is returned.

Returns

read_files(*mode='r'*, *text=False*)

open_file(*filename=None*, *mode='r'*)

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for NamedFileCollection

write_file(*filename*, *data*, *text=False*)
If text=True, the data is expected to be unicode and is encoded as utf-8. Otherwise, data should be a bytes object.

file_written(*filename*)
Mark the given file as having been written, if write_file() was not used to write it.

open_file(*filename=None*)

get_absolute_path(*filename=None*)

absolute_paths

metadata_defaults = {}

writer_param_defaults = {}

class NamedFile(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

Like NamedFileCollection, but always has exactly one file.

The filename is given as the *filename* datatype option, which can also be given as the first init arg: *NamedFile("myfile.txt")*.

Since NamedFile is a subtype of NamedFileCollection, it also has a "filenames" option. It is ignored if the *filename* option is given, and otherwise must have exactly one item.

datatype_name = 'named_file'

datatype_options = {'filename': {'help': "The file's name"}, 'filenames': {'default

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for NamedFile

process_setup()
Do any processing of the setup object (e.g. retrieving values and setting attributes on the reader) that should be done when the reader is instantiated.

absolute_path

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for NamedFile.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *NamedFile.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for NamedFile

write_file(*data*, *text=False*)
If text=True, the data is expected to be unicode and is encoded as utf-8. Otherwise, data should be a bytes object.

absolute_path

metadata_defaults = {}

writer_param_defaults = {}

class FilesInput(*min_files=1*)
Bases: *pimlico.datatypes.base.DynamicInputDatatypeRequirement*

datatype_doc_info = 'A file collection containing at least one file (or a given specif

check_type(*supplied_type*)

FileInput
alias of *pimlico.datatypes.files.FilesInput*

class TextFile(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFile*

Simple dataset containing just a single utf-8 encoded text file.

datatype_name = 'text_document'

datatype_options = {'filename': {'default': 'data.txt', 'help': "The file's name. Ty

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for TextFile

read_file(*filename=None*, *mode='r'*, *text=False*)
Read a file from the collection.

Parameters

- **filename** – string filename, which should be one of the filenames specified for this collection; or an integer, in which case the ith file in the collection is read. If not given, the first file is read
- **mode** –
- **text** – if True, the file is treated as utf-8-encoded text and a unicode object is returned. Otherwise, a bytes object is returned.

Returns

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for TextFile.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *TextFile.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for TextFile

metadata_defaults = {}

writer_param_defaults = {}

write_file(*data*, *text=False*)
If text=True, the data is expected to be unicode and is encoded as utf-8. Otherwise, data should be a bytes object.

gensim

class GensimLdaModel(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Storage of trained Gensim LDA models.

Depends on Gensim (and thereby also in Python 3), since we use Gensim to store and load the models.

datatype_name = 'lda_model'

datatype_supports_python2 = False

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

run_browser(*reader*, *opts*)
Browse the LDA model simply by printing out all its topics.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for GensimLdaModel

load_model()

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for GensimLdaModel.Reader

data_ready(*path*)
Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()
Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()
Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)
Instantiate a reader using this setup.
Parameters
- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type
alias of *GensimLdaModel.Reader*

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.
Returns True if the reader's ready to be instantiated, False otherwise

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for GensimLdaModel

required_tasks = ['model']

write_model(*model*)

metadata_defaults = {}

writer_param_defaults = {}

class TopicsTopWords(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Stores a list of the top words for each topic of a topic model.

For some evaluations (like coherence), this is all the information that is needed about a model. This datatype can be extracted from various topic model types, so that they can all be evaluated using the same evaluation modules.

datatype_name = 'topics_top_words'

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for TopicsTopWords

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for TopicsTopWords.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *TopicsTopWords.Reader*

topics_words

num_topics

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for TopicsTopWords

required_tasks = ['topics.tsv']

write_topics_words(*topics_words*)

Parameters topics_words – list of topic, where each topic is a list of words, with the top weighted word first

metadata_defaults = {}

writer_param_defaults = {}

run_browser(*reader*, *opts*)
Launches a browser interface for reading this datatype, browsing the data provided by the given reader.

Not all datatypes provide a browser. For those that don't, this method should raise a NotImplementedError.

opts provides the argparser options from the command line.

This tool used to be only available for iterable corpora, but now it's possible for any datatype to provide a browser. IterableCorpus provides its own browser, as before, which uses one of the data point type's formatters to format documents.

keras

Datatypes for storing and loading Keras models.

class KerasModel(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Datatype for both types of Keras models, stored using Keras' own storage mechanisms. This uses Keras' method of storing the model architecture as JSON and stores the weights using hdf5.

datatype_name = 'keras_model'

custom_objects = {}

datatype_supports_python2 = True

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You

might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for KerasModel

get_custom_objects()

load_model()

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for KerasModel.Reader

data_ready(*path*)
Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()
Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()
Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)
Instantiate a reader using this setup.
Parameters
- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed

outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type
alias of *KerasModel.Reader*

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.
Returns True if the reader's ready to be instantiated, False otherwise

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for KerasModel

required_tasks = ['architecture', 'weights']

weights_filename

write_model(*model*)

write_architecture(*model*)

write_weights(*model*)

metadata_defaults = {}

writer_param_defaults = {}

class KerasModelBuilderClass(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

An alternative way to store Keras models.

Create a class whose init method build the model architecture. It should take a kwarg called *build_params*, which is a JSON-encodable dictionary of parameters that determine how the model gets build (hyperparameters). When you initialize your model for training, create this hyperparameter dictionary and use it to instantiate the model class.

Use the KerasModelBuilderClassWriter to store the model during training. Create a writer, then start model training, storing the weights to the filename given by the *weights_filename* attribute of the writer. The hyperparameter dictionary will also be stored.

The writer also stores the fully-qualified path of the model-builder class. When we read the datatype and want to rebuild the model, we import the class, instantiate it and then set its weights to those we've stored.

The model builder class must have the model stored in an attribute *model*.

datatype_name = 'keras_model_builder_class'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.base.Reader

Reader class for KerasModelBuilderClass

weights_filename

load_build_params()

create_builder_class(*override_params=None*)

load_model(*override_params=None*)
Instantiate the model builder class with the stored parameters and set the weights on the model to those stored.

Returns model builder instance (keras model in attribute *model*

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.base.Setup

Setup class for KerasModelBuilderClass.Reader

data_ready(*path*)
Check whether the data at the given path is ready to be read using this type of reader. It may be called several times with different possible base dirs to check whether data is available at any of them.

Often you will override this for particular datatypes to provide special checks. You may (but don't have to) check the setup's parent implementation of *data_ready()* by calling *super(MyDatatype.Reader.Setup, self).data_ready(path)*.

The base implementation just checks whether the data dir exists. Subclasses will typically want to add their own checks.

get_base_dir()
Returns the first of the possible base dir paths at which the data is ready to read. Raises an exception if none is ready. Typically used to get the path from the reader, once we've already confirmed that at least one is available.

get_data_dir()
Returns the path to the data dir within the base dir (typically a dir called "data")

get_reader(*pipeline*, *module=None*)
Instantiate a reader using this setup.
Parameters
- **pipeline** – currently loaded pipeline
- **module** – (optional) module name of the module by which the datatype has been loaded. Used for producing intelligible error output

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

read_metadata(*base_dir*)
Read in metadata for a dataset stored at the given path. Used by readers and rarely needed outside them. It may sometimes be necessary to call this from *data_ready()* to check that required metadata is available.

reader_type
alias of *KerasModelBuilderClass.Reader*

ready_to_read()
Check whether we're ready to instantiate a reader using this setup. Always called before a reader is instantiated.

Subclasses may override this, but most of the time you won't need to. See *data_ready()* instead.
Returns True if the reader's ready to be instantiated, False otherwise

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for KerasModelBuilderClass

required_tasks = ['architecture', 'weights']

write_weights(*model*)

metadata_defaults = {}

writer_param_defaults = {}

plotting

class PlotOutput(**args*, ***kwargs*)

Bases: *pimlico.datatypes.files.NamedFileCollection*

Output from matplotlib plotting.

Contains the dataset being plotted, a script to build the plot, and the output PDF.

datatype_supports_python2 = True

class Writer(**args*, ***kwargs*)

Bases: pimlico.datatypes.files.Writer

Writes out source data, a Python script for the plotting using Matplotlib and a PDF of the resulting plot, if the script completes successfully.

This approach means that a plot is produced immediately, but can easily be tweaked and customized for later use elsewhere by copying and editing the Python plotting script.

Use `writer.write_file("data.csv", text=True)` to write the source data and `writer.write_file("plot.py", text=True)` to write the plotting script, which should output a file `plot.pdf`. Then call `writer.plot()` to execute the script. If this fails, at least the other files are there so the user can correct the errors and use them if they want.

plot()

Runs the plotting script. Errors are not caught, so if there's a problem in the script they'll be raised.

data_path

code_path

plot_path

metadata_defaults = {}

writer_param_defaults = {}

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)

Bases: pimlico.datatypes.base.Reader

Reader class for NamedFileCollection

class Setup(*datatype*, *data_paths*)

Bases: pimlico.datatypes.base.Setup

Setup class for NamedFileCollection.Reader

get_required_paths()

May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type

alias of NamedFileCollection.Reader

absolute_filenames
For backwards compatibility: use absolute_paths by preference

absolute_paths

get_absolute_path(*filename*)

open_file(*filename=None*, *mode='r'*)

process_setup()
Do any processing of the setup object (e.g. retrieving values and setting attributes on the reader) that should be done when the reader is instantiated.

read_file(*filename=None*, *mode='r'*, *text=False*)
Read a file from the collection.

Parameters

- **filename** – string filename, which should be one of the filenames specified for this collection; or an integer, in which case the ith file in the collection is read. If not given, the first file is read
- **mode** –
- **text** – if True, the file is treated as utf-8-encoded text and a unicode object is returned. Otherwise, a bytes object is returned.

Returns

read_files(*mode='r'*, *text=False*)

results

class NumericResult(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Simple datatype to contain a numeric value and a label, representing the result of some process, such as evaluation of a model on a task.

Write using `writer.write(label, value)`. The label must be a string, identifying what the result is, e.g. "f-score". The value can be any JSON-serializable type, e.g. int or float.

For example, allows results to be plotted by passing them into a graph plotting module.

datatype_name = 'numeric_result'

datatype_supports_python2 = True

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: `pimlico.datatypes.base.Reader`

Reader class for NumericResult

class Setup(*datatype*, *data_paths*)
Bases: `pimlico.datatypes.base.Setup`

Setup class for NumericResult.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *NumericResult.Reader*

data

label

value

class Writer(*datatype*, *base_dir*, *pipeline*, *module=None*, ***kwargs*)
Bases: pimlico.datatypes.base.Writer

Writer class for NumericResult

required_tasks = ['data']

write(*label*, *value*)

metadata_defaults = {}

writer_param_defaults = {}

sklearn

class SklearnModel(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFile*

Saves and loads scikit-learn models using the library's joblib functions.

See the sklearn docs for more details

datatype_name = 'sklearn_model'

datatype_supports_python2 = True

get_software_dependencies()
Get a list of all software required to **read** this datatype. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

Note that there may be different software dependencies for **writing** a datatype using its *Writer*. These should be specified using *get_writer_software_dependencies()*.

class Reader(*datatype*, *setup*, *pipeline*, *module=None*)
Bases: pimlico.datatypes.files.Reader

Reader class for SklearnModel

load_model()

class Setup(*datatype*, *data_paths*)
Bases: pimlico.datatypes.files.Setup

Setup class for SklearnModel.Reader

get_required_paths()
May be overridden by subclasses to provide a list of paths (absolute, or relative to the data dir) that must exist for the data to be considered ready.

reader_type
alias of *SklearnModel.Reader*

class Writer(**args*, ***kwargs*)
Bases: pimlico.datatypes.files.Writer

Writer class for SklearnModel

save_model(*model*)

metadata_defaults = {}

writer_param_defaults = {}

Module contents

load_datatype(*path*, *options={}*)
Try loading a datatype class for a given path. Raises a DatatypeLoadError if it's not a valid datatype path. Also looks up class names of builtin datatypes and datatype names.

Options are unprocessed strings that will be processed using the datatype's option definitions.

test

Submodules

pipeline

suite

Module contents

utils

Subpackages

docs

Submodules

apiheaders

Tiny script to replace the headers in the API docs after they've been built using sphinx-apidoc.

See: https://stackoverflow.com/questions/25276164/sphinx-apidoc-dont-print-full-path-to-packages-and-modules

Only works in Python 3: it's assume docs are built in Python 3.

commandgen

Tool to generate Pimlico command docs. Based on Sphinx's apidoc tool.

generate_docs(*output_dir*)
Generate RST docs for Pimlico commands and output to a directory.

generate_docs_for_command(*command_cls*, *output_dir*)

generate_contents_page(*commands*, *command_descs*, *output_dir*)

cap_first(*txt*)

strip_common_indent(*code*)

examplegen

Tool to generate Pimlico docs for example config files.

Each example config file is (for now) just shown in full in the docs.

build_example_config_doc(*base_path*, *rel_path*)

build_index(*generated*, *output_dir*)

build_example_config_docs(*example_config_dir*, *output_dir*)

modulegen

Tool to generate Pimlico module docs. Based on Sphinx's apidoc tool.

It is assumed that this script will be run using Python 3. Although it has a basic Python 2 compatibility, it's not really intended for Python 2 use. Modules that are marked as still awaiting update to the new datatypes system will now not be imported at all, since they are typically not Python 3 compatible (due to their use of `old_datatypes`, which has not been updated to Python 3).

generate_docs_for_pymod(*module*, *output_dir*, *test_refs={}*, *example_refs={}*)
Generate RST docs for Pimlico modules on a given Python path and output to a directory.

generate_docs_for_pimlico_mod(*module_path*, *output_dir*, *submodules=[]*, *test_refs={}*, *example_refs={}*)

input_datatype_list(*types*, *context=None*, *no_warn=False*)

input_datatype_text(*datatype*, *context=None*, *no_warn=False*)

output_datatype_text(*datatype*, *context=None*, *no_warn=False*)

datatype_to_link(*datatype_inst*)

generate_contents_page(*modules*, *output_dir*, *index_name*, *title*, *content*)

generate_example_config(*info*, *input_types*, *module_path*, *minimal=False*)
Generate a string containing an example of how to configure the given module in a pipeline config file. Where possible, uses default values for options, or values appropriate to the type, and dummy input names.

indent(*spaces*, *text*)

rest

make_table(*grid*, *header=None*)

table_div(*col_widths*, *header_flag=False*)

normalize_cell(*string*, *length*)

format_heading(*level*, *text*, *escape=True*)
Create a heading of <level> [1, 2 or 3 supported].

testgen

Module contents

trim_docstring(*docstring*)

pimarc

Submodules

index

class PimarcIndex
Bases: object

Simple index to accompany a Pimarc, stored along with the *.prc* file as a *.prci* file. Provides a list of the filenames in the archive, along with the starting byte of the file's metadata and data.

filenames is an OrderedDict mapping filename -> (metadata start byte, data start byte).

get_metadata_start_byte(*filename*)

get_data_start_byte(*filename*)

keys()

append(*filename*, *metadata_start*, *data_start*)

static load(*filename*)

save(*path*)

class PimarcIndexAppender(*store_path*, *mode='w'*)
Bases: object

Class for writing out a Pimarc index as each file is added to the archive. This is used by the Pimarc writer, instead of creating a PimarcIndex and calling *save()*, so that the index is always kept up to date with what's in the archive.

Mode may be *"w"* to write a new index or *"a"* to append to an existing one.

append(*filename*, *metadata_start*, *data_start*)

close()

flush()

reindex(*pimarc_path*)
Rebuild the index of a Pimarc archive from its data file (.prc).

Stores the new index in the correct location (.prci), overwriting any existing index.

Parameters **pimarc_path** – path to the .prc file

Returns the PimarcIndex

check_index(*pimarc_path*)
Check through a Pimarc file together with its index to identify any places where the index does not match the archive contents.

Useful for debugging writing/reading code.

exception IndexCheckFailed
Bases: Exception

exception FilenameNotInArchive(*filename*)
Bases: Exception

exception DuplicateFilename(*filename*)
Bases: Exception

exception IndexWriteError
Bases: Exception

reader

class PimarcReader(*archive_filename*)
Bases: object

The Pimlico Archive format: read-only archive.

close()

read_file(*filename*)
Load a file. Same as *reader[filename]*

iter_filenames()
Iterate over just the filenames in the archive, without further metadata or file data. Fast for Pimarc, as the index is fully loaded into memory.

iter_metadata()
Iterate over all files in the archive, yielding just the metadata, skipping over the data.

iter_files(*skip=None*, *start_after=None*)
Iterate over files, together with their JSON metadata, which includes their name (as "name").

Parameters

- **start_after** – skips all files before that with the given name, which is expected to be in the archive
- **skip** – skips over the first portion of the archive, until this number of documents have been seen. Ignored is start_after is given.

read_doc_from_pimarc(*archive_filename*, *metadata_start_byte*)
Read a single file's metadata and file data from a given start point in the archive. This can be useful if you know the start point and don't want to read in the whole index for an archive.

Parameters

- **archive_filename** – path to archive file
- **metadata_start_byte** – byte from which metadata starts

Returns tuple (metadata, raw file data)

read_doc_from_pimarc_file(*archive_file*, *metadata_start_byte*)
Same as *read_doc_from_pimarc*, but operates on an already-opened archive file.

Parameters

- **archive_file** – file-like object
- **metadata_start_byte** – byte from which metadata starts

Returns tuple (metadata, raw file data)

metadata_decode_decorator(*fn*)

class PimarcFileMetadata(*raw_data*)
Bases: `dict`

Simple wrapper around the JSON-encoded metadata associated with a file in a Pimarc archive. When the metadata is loaded, the raw bytes data is wrapped in an instance of PimarcFileMetadata, so that it can be easily decoded when needed, but avoiding decoding all metadata, which might not ever be needed.

You can simply use the object as if it is a dict and it will decode the JSON data the first time you try accessing it. You can also call *dict(obj)* to get a plain dict instead.

decode()

keys(**args*, ***kwargs*)

values(**args*, ***kwargs*)

items(**args*, ***kwargs*)

exception StartAfterFilenameNotFound
Bases: `KeyError`

tar

Wrapper around tar reader, to provide the same interface as Pimarc.

This means we can deprecate the use of tar files, but keep backwards compatibility for a time, whilst moving over to direct use of Pimarc objects.

class PimarcTarBackend(*archive_filename*)
Bases: `object`

open()

close()

iter_filenames()
Just iterate over the filenames (decoded if necessary). Used to create metadata, check for file existence, etc.

Not as fast as with Pimarc, as we need to pass over the whole archive file to read all the names.

iter_metadata()
Iterate over all files in the archive, yielding just the metadata, skipping over the data.

iter_files(*skip=None*, *start_after=None*)
Iterate over files, together with their JSON metadata, which includes their name (as "name").

Parameters

- **start_after** – skips all files before that with the given name, which is expected to be in the archive
- **skip** – skips over the first portion of the archive, until this number of documents have been seen. Ignored is start_after is given.

tools

Command-line tools for manipulating Pimarcs.

list_files(*opts*)

extract_file(*opts*)

append_file(*opts*)

from_tar(*opts*)

reindex_pimarcs(*opts*)

check_pimarcs(*opts*)

remove(*opts*)

no_subcommand(*opts*)

run()

utils

writer

class PimarcWriter(*archive_filename*, *mode='w'*)

Bases: `object`

The Pimlico Archive format: writing new archives or appending existing ones.

static delete(*archive_filename*)

Delete all files associated with the given archive. At the moment, this is just the archive file itself and the associated index.

close()

write_file(*data*, *name=None*, *metadata=None*)

Append a write to the end of the archive. The metadata should be a dictionary that can be encoded as JSON (which is how it will be stored). The data should be a bytes object.

If you want to write text files, you should encode the text as UTF-8 to get a bytes object and write that.

Setting *name=X* is simply a shorthand for setting *metadata["name"]=X*. Either *name* or a metadata dict including the *name* key is required.

flush()

Flush the archive's data out to disk, archive and index.

exception MetadataError

Bases: `Exception`

Module contents

The Pimlico Archive format

Implementation of a simple multi-file archive format, somewhat like tar.

Pimlico multi-file datasets currently use tar to store many files in one archive. This was attractive because of its simplicity and the fact that the files can be iterated over in order efficiently. However, tar is an old format and has certain quirks. The biggest downside is that random access (reading files not in the order stored or jumping into the middle of an archive) is very slow.

The Pimlico Archive format (prc) aims to be a very simple generic archive format. It has the same property as tars that it is fast to iterate over files in order. But it also stores an index that can be loaded into memory to make it quick to jump into the archive and potentially access the files in a random order.

It stores very little information about the files. In this sense, it is simpler than tar. It does not store, for example, file timestamps or permissions, since we do not need these things for documents in a Pimlico corpus. It does, however, have a generic JSON metadata dictionary for each file, so metadata like this can be stored as necessary.

Pimarcs do not store any directory structures, just a flat collection of files. This is all that is needed for storing Pimlico datasets, so it's best for this purpose to keep the format as simple as possible.

Iterating over files in order is still likely to be substantially faster than random access (depending on the underlying storage), so it is recommended to add files to the archive in the sequential order that they are used in. This is the typical use case in Pimlico: a dataset is created in order, one document at a time, and stored iteratively. Then another module reads and processes those documents in the same order.

In keeping with this typical use case in Pimlico, a Pimarc can be opened for reading only, writing only (new archive) or appending, just like normal files. You cannot, for example, open an archive and move files around, or delete a file. To do these things, you must read in an archive using a reader and write out a new, modified one using a writer.

Restrictions on filenames: Filenames may use any unicode characters, excluding EOF, newline and tab.

The standard filename for a Pimarc file is *.prc*. This file contains the archive's data. A second file is always stored in the same location, with an identical filename, except the extension *.prci*.

Some basic command-line utilities for working with Pimarc archives are provided. Run *pimlico.utils.pimarc* with one of the various sub-commands.

open_archive(*path*, *mode='r'*)

Submodules

communicate

timeout_process(*proc*, *timeout*)

Context manager for use in a *with* statement. If the with block hasn't completed after the given number of seconds, the process is killed.

Parameters **proc** – process to kill if timeout is reached before end of block

Returns

terminate_process(*proc*, *kill_time=None*)

Ends a process started with subprocess. Tries killing, then falls back on terminating if it doesn't work.

Parameters

- **kill_time** – time to allow the process to be killed before falling back on terminating
- **proc** – Popen instance

Returns

class StreamCommunicationPacket(*data*)
Bases: object

length

encode()

static read(*stream*)

exception StreamCommunicationError
Bases: Exception

core

multiwith(**managers*)
Taken from contextlib's nested(). We need the variable number of context managers that this function allows.

is_identifier(*ident*)
Determines if string is valid Python identifier.

remove_duplicates(*lst*, *key=<function <lambda>>*)
Remove duplicate values from a list, keeping just the first one, using a particular key function to compare them.

infinite_cycle(*iterable*)
Iterate infinitely over the given iterable.

Watch out for calling this on a generator or iter: they can only be iterated over once, so you'll get stuck in an infinite loop with no more items yielded once you've gone over it once.

You may also specify a callable, in which case it will be called each time to get a new iterable/iterator. This is useful in the case of generator functions.

Parameters **iterable** – iterable or generator to loop over indefinitely

import_member(*path*)
Import a class, function, or other module member by its fully-qualified Python name.

Parameters **path** – path to member, including full package path and class/function/etc name

Returns cls

split_seq(*seq*, *separator*, *ignore_empty_final=False*)
Iterate over a sequence and group its values into lists, separated in the original sequence by the given value. If *on* is callable, it is called on each element to test whether it is a separator. Otherwise, elements that are equal to *on* a treated as separators.

Parameters

- **seq** – sequence to divide up
- **separator** – separator or separator test function
- **ignore_empty_final** – by default, if there's a separator at the end, the last sequence yielded is empty. If ignore_empty_final=True, in this case the last empty sequence is dropped

Returns iterator over subsequences

split_seq_after(*seq*, *separator*)
Somewhat like split_seq, but starts a new subsequence after each separator, without removing the separators. Each subsequence therefore ends with a separator, except the last one if there's no separator at the end.

Parameters

- **seq** – sequence to divide up
- **separator** – separator or separator test function

Returns iterator over subsequences

chunk_list(*lst*, *length*)
Divides a list into chunks of max *length* length.

class cached_property(*func*)
Bases: `object`

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property.

Often useful in Pimlico datatypes, where it can be time-consuming to load data, but we can't do it once when the datatype is first loaded, since the data might not be ready at that point. Instead, we can access the data, or particular parts of it, using properties and easily cache the result.

Taken from: https://github.com/bottlepy/bottle

email

Email sending utilities

Configure email sending functionality by adding the following fields to your Pimlico local config file:

email_sender From-address for all sent emails

email_recipients To-addresses, separated by commas. All notification emails will be sent to all recipients

email_host (optional) Hostname of your SMTP server. Defaults to *localhost*

email_username (optional) Username to authenticate with your SMTP server. If not given, it is assumed that no authentication is required

email_password (optional) Password to authenticate with your SMTP server. Must be supplied if *username* is given

class EmailConfig(*sender=None*, *recipients=None*, *host=None*, *username=None*, *password=None*)
Bases: `object`

classmethod from_local_config(*local_config*)

send_pimlico_email(*subject*, *content*, *local_config*, *log*)
Primary method for sending emails from Pimlico. Tries to send an email with the given content, using the email details found in the local config. If something goes wrong, an error is logged on the given log.

Parameters

- **subject** – email subject
- **content** – email text (may be unicode)
- **local_config** – local config dictionary
- **log** – logger to log errors to (and info if the sending works)

send_text_email(*email_config*, *subject*, *content=None*)

exception EmailError
Bases: `Exception`

filesystem

dirsize(*path*)
Recursively compute the size of the contents of a directory.

Parameters **path** –

Returns size in bytes

format_file_size(*bytes*)

copy_dir_with_progress(*source_dir*, *target_dir*, *move=False*)
Utility for moving/copying a large directory and displaying a progress bar showing how much is copied.

Note that the directory is first copied, then the old directory is removed, if move=True.

Parameters

- **source_dir** –
- **target_dir** –

Returns

move_dir_with_progress(*source_dir*, *target_dir*)

new_filename(*directory*, *initial_filename='tmp_file'*)
Generate a filename that doesn't already exist.

retry_open(*filename, errnos=[13], retry_schedule=[2, 10, 30, 120, 300], **kwargs*)
Try opening a file, using the builtin open() function (Py3, or io.open on Py2). If an IOError is raised and its *errno* is in the given list, wait a moment then retry. Keeps doing this, waiting a bit longer each time, hoping that the problem will go away.

Once too many attempts have been made, outputs a message and waits for user input. This means the user can fix the problem (e.g. renew credentials) and pick up where execution left off. If they choose not to, the original error will be raised

Default list of errnos is just *[13]* – permission denied.

Use *retry_schedule* to customize the lengths of time waited between retries. Default: 2s, 10s, 30s, 2m, 5m, then give up.

Additional kwargs are pass on to *open()*.

extract_from_archive(*archive_filename*, *members*, *target_dir*, *preserve_dirs=True*)
Extract a file or files from an archive, which may be a tarball or a zip file (determined by the file extension).

extract_archive(*archive_filename*, *target_dir*, *preserve_dirs=True*)
Extract all files from an archive, which may be a tarball or a zip file (determined by the file extension).

format

multiline_tablate(*table*, *widths*, ***kwargs*)

title_box(*title_text*)
Make a nice big pretty title surrounded by a box.

jupyter

get_pipeline()

linguistic

strip_punctuation(*s*, *split_words=True*)

logging

get_console_logger(*name*, *debug=False*)
Convenience function to make it easier to create new loggers.

Parameters

- **name** – logging system logger name
- **debug** – whether to use DEBUG level. By default, uses INFO

Returns

network

get_unused_local_port()
Find a local port that's not currently being used, which we'll be able to bind a service to once this function returns.

get_unused_local_ports(*n*)
Find a number of local ports not currently in use. Binds each port found before looking for the next one. If you just called get_unused_local_port() multiple times, you'd get to same answer coming back.

pipes

qget(*queue*, **args*, ***kwargs*)
Wrapper that calls the get() method of a queue, catching EINTR interrupts and retrying. Recent versions of Python have this built in, but with earlier versions you can end up having processes die while waiting on queue output because an EINTR has received (which isn't necessarily a problem).

Parameters

- **queue** –
- **args** – args to pass to queue's get()
- **kwargs** – kwargs to pass to queue's get()

Returns

class OutputQueue(*out*)
Bases: `object`

Direct a readable output (e.g. pipe from a subprocess) to a queue. Returns the queue. Output is added to the queue one line at a time. To perform a non-blocking read call get_nowait() or get(timeout=T)

get_nowait()

get(*timeout=None*)

get_available()
Don't block. Just return everything that's available in the queue.

pos

pos_tag_to_ptb(*tag*)
see :doc:pos_pos_tags_to_ptb

pos_tags_to_ptb(*tags*)
Takes a list of POS tags and checks they're all in the PTB tagset. If they're not, tries mapping them according to CCGBank's special version of the tagset. If that doesn't work, raises a NonPTBTagError.

exception NonPTBTagError
Bases: `Exception`

probability

limited_shuffle(*iterable*, *buffer_size*, *rand_generator=None*)
Some algorithms require the order of data to be randomized. An obvious solution is to put it all in a list and shuffle, but if you don't want to load it all into memory that's not an option. This method iterates over the data, keeping a buffer and choosing at random from the buffer what to put next. It's less shuffled than the simpler solution, but limits the amount of memory used at any one time to the buffer size.

limited_shuffle_numpy(*iterable*, *buffer_size*, *randint_buffer_size=1000*)
Identical behaviour to *limited_shuffle()*, but uses Numpy's random sampling routines to generate a large number of random integers at once. This can make execution a bit bursty, but overall tends to speed things up, as we get the random sampling over in one big call to Numpy.

batched_randint(*low*, *high=None*, *batch_size=1000*)
Infinite iterable that produces random numbers in the given range by calling Numpy now and then to generate lots of random numbers at once and then yielding them one by one. Faster than sampling one at a time.

Parameters

- **a** – lowest number in range
- **b** – highest number in range
- **batch_size** – number of ints to generate in one go

sequential_document_sample(*corpus*, *start=None*, *shuffle=None*, *sample_rate=None*)
Wrapper around a `pimlico.datatypes.tar.TarredCorpus` to draw infinite samples of documents from the corpus, by iterating over the corpus (looping infinitely), yielding documents at random. If *sample_rate* is given, it should be a float between 0 and 1, specifying the rough proportion of documents to sample. A lower value spreads out the documents more on average.

Optionally, the samples are shuffled within a limited scope. Set *shuffle* to the size of this scope (higher will shuffle more, but need to buffer more samples in memory). Otherwise (*shuffle=0*), they will appear in the order they were in the original corpus.

If *start* is given, that number of documents will be skipped before drawing any samples. Set *start=0* to start at the beginning of the corpus. By default (*start=None*) a random point in the corpus will be skipped to before beginning.

sequential_sample(*iterable*, *start=0*, *shuffle=None*, *sample_rate=None*)
Draw infinite samples from an iterable, by iterating over it (looping infinitely), yielding items at random. If *sample_rate* is given, it should be a float between 0 and 1, specifying the rough proportion of documents to sample. A lower value spreads out the documents more on average.

Optionally, the samples are shuffled within a limited scope. Set *shuffle* to the size of this scope (higher will shuffle more, but need to buffer more samples in memory). Otherwise (*shuffle=0*), they will appear in the order they were in the original corpus.

If *start* is given, that number of documents will be skipped before drawing any samples. Set *start=0* to start at the beginning of the corpus. Note that setting this to a high number can result in a slow start-up, if iterating over the items is slow.

Note: If you're sampling documents from a *TarredCorpus*, it's better to use `sequential_document_sample()`, since it makes use of *TarredCorpus*'s built-in features to do the skipping and sampling more efficiently.

---

subsample(*iterable*, *sample_rate*)
Subsample the given iterable at a given rate, between 0 and 1.

progress

get_progress_bar(*maxval*, *counter=False*, *title=None*, *start=True*)
Simple utility to build a standard progress bar, so I don't have to think about this each time I need one. Starts the progress bar immediately.

start is no longer used, included only for backwards compatibility.

get_open_progress_bar(*title=None*)
Builds a standard progress bar for the case where the total length (max value) is not known, i.e. an open-ended progress bar.

class SafeProgressBar(*maxval=None*, *widgets=None*, *term_width=None*, *poll=1*, *left_justify=True*, *fd=None*)
Bases: `progressbar.progressbar.ProgressBar`

Override basic progress bar to wrap update() method with a couple of extra features.

1. You don't need to call start() – it will be called when the first update is received. This is good for processes that have a bit of a start-up lag, or where starting to iterate might generate some other output.
2. An error is not raised if you update with a value higher than maxval. It's the most annoying thing ever if you run a long process and the whole thing fails near the end because you slightly miscalculated maxval.

Initializes a progress bar with sane defaults.

update(*value=None*)
Updates the ProgressBar to a new value.

increment()

class DummyFileDescriptor
Bases: `object`

Passed in to ProgressBar instead of a file descriptor (e.g. stderr) to ensure that nothing gets output.

read(*size=None*)

readLine(*size=None*)

write(*s*)

close()

class NonOutputtingProgressBar(**args*, ***kwargs*)
Bases: *pimlico.utils.progress.SafeProgressBar*

Behaves like ProgressBar, but doesn't output anything.

class LittleOutputtingProgressBar(**args*, ***kwargs*)
Bases: *pimlico.utils.progress.SafeProgressBar*

Behaves like ProgressBar, but doesn't output much. Instead of constantly redrawing the progress bar line, it outputs a simple progress message every time it hits the next 10% mark.

If running on a terminal, this will update the line, as with a normal progress bar. If piping to a file, this will just print a new line occasionally, so won't fill up your file with thousands of progress updates.

start()
Starts measuring time, and prints the bar at 0%.

It returns self so you can use it like this: >>> pbar = ProgressBar().start() >>> for i in range(100): ... # do something ... pbar.update(i+1) ... >>> pbar.finish()

finish()
Puts the ProgressBar bar in the finished state.

slice_progress(*iterable*, *num_items*, *title=None*)

class ProgressBarIter(*iterable*, *title=None*)
Bases: object

strings

truncate(*s*, *length*, *ellipsis='...'*)

similarities(*targets*, *reference*)
Compute string similarity of each of a list of targets to a given reference string. Uses *difflib.SequenceMatcher* to compute similarity.

Parameters

- **reference** – compare all strings to this one
- **targets** – list of targets to measure similarity of

Returns list of similarity values

sorted_by_similarity(*targets*, *reference*)
Return target list sorted by similarity to the reference string. See :func:similarities for similarity measurement.

system

Lowish-level system operations

set_proc_title(*title*)
Tries to set the current process title. This is very system-dependent and may not always work.

If it's available, we use the *setproctitle* package, which is the most reliable way to do this. If not, we try doing it by loading libc and calling *prctl* ourselves. This is not reliable and only works on Unix systems. If neither of these works, we give up and return False.

If you want to increase the chances of this working (e.g. your process titles don't seem to be getting set by Pimlico and you'd like them to), try installing *setproctitle*, either system-wide or in Pimlico's virtualenv.

@return: True if the process succeeds, False if there's an error

timeout

timeout (*func*, *args=()*, *kwargs={}*, *timeout_duration=1*, *default=None*)

urwid

Some handy Urwid utilities.

Take care only to import this where we already have a dependency on Urwid, e.g. in the browser implementation modules.

Some of these are taken pretty exactly from Urwid examples.

Todo: Not got these things working yet, but they'll be useful in the long run

---

exception DialogExit
Bases: `Exception`

class DialogDisplay (*original_widget*, *text*, *height=0*, *width=0*, *body=None*)
Bases: `urwid.wimp.PopUpLauncher`

palette = [('body', 'black', 'light gray', 'standout'), ('border', 'black', 'dark blue

add_buttons (*buttons*)

button_press (*button*)

on_exit (*exitcode*)

class ListDialogDisplay (*original_widget*, *text*, *height*, *width*, *constr*, *items*, *has_default*)
Bases: *pimlico.utils.urwid.DialogDisplay*

unhandled_key (*size*, *k*)

on_exit (*exitcode*)
Print the tag of the item selected.

msgbox (*original_widget*, *text*, *height=0*, *width=0*)

options_dialog (*original_widget*, *text*, *options*, *height=0*, *width=0*, **items*)

yesno_dialog (*original_widget*, *text*, *height=0*, *width=0*, **items*)

varint

Varint encoder/decoder

Implementation of a variable-length integer encoding scheme.

Based on implementation by Peter Ruibal: https://github.com/fmoo/python-varint

It's copied here so we can use it stably without adding a dependency.

License: Since this is copied from someone else's code, its license is that of the original code, the MIT license. See LICENSE below for details.

Varints are a common encoding for variable-length integer data, used in libraries such as sqlite, protobuf, v8, and more.

Here's a quick and dirty module to help avoid reimplementing the same thing over and over again.

encode(*number*)
Pack *number* into varint bytes

decode_stream(*stream*)
Read a varint from *stream*

decode_bytes(*buf*)
Read a varint from from *buf* bytes

web

download_file(*url*, *target_file*, *headers=None*)
Now just an alias for urllib.urlretrieve()

Module contents

Submodules

cfg

Global config

Various global variables. Access as follows:

from pimlico import cfg

Set global config parameter cfg.parameter = "Value" # Use parameter print cfg.parameter

There are some global variables in `pimlico` (in the __init__.py) that probably should be moved here, but I'm leaving them for now. At the moment, none of those are ever written from outside that file (i.e. think of them as constants, rather than config), so the only reason to move them is to keep everything in one place.

Module contents

The Pimlico Processing Toolkit (PIpelined Modular LInguistic COrpus processing) is a toolkit for building pipelines made up of linguistic processing tasks to run on large datasets (corpora). It provides a wrappers around many existing, widely used NLP (Natural Language Processing) tools.

install_core_dependencies()

get_jupyter_pipeline()
Special function to get access to a currently loaded pipeline from a Jupyter notebook.

1.6 Module test pipelines

Test pipelines provide a special sort of unit testing for Pimlico.

Pimlico is distributed with a set of test pipeline config files, each just a small pipeline with a couple of modules in it. Each is designed to test the use of a particular one of Pimlico's builtin module types, or some combination of a smaller number of them.

1.6.1 Available pipelines

lda_train

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Train an LDA model on a pre-prepared word-ID corpus
#
# For a fuller example (on which this test is based), see
# :doc:`the topic model training example </example_config/topic_modelling.train_tms>`.

[pipeline]
name=lda_train
release=latest

# Load word IDs
[ids]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=IntegerListsDocumentType
dir=%(test_data_dir)s/datasets/corpora/ids_ubuntu

# Load vocabulary
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab_ubuntu

# First train a plain LDA model using Gensim
[lda]
type=pimlico.modules.gensim.lda
input_vocab=vocab
input_corpus=ids
tfidf=T
# Small number of topics: you probably want more in practice
num_topics=5
passes=10
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.gensim.lda*

lda_coherence

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Compute coherence of trained topics over a reference dataset
#
# The topic model was trained by the LDA training test pipeline and
# its topics' top words have been extracted by the top words test pipeline.
#
# The "reference set" is the small test tokenized dataset.

[pipeline]
name=lda_coherence
release=latest

# Load top words for the topics
[top_words]
type=pimlico.datatypes.gensim.TopicsTopWords
dir=%(test_data_dir)s/datasets/gensim/lda_top_words

# Load vocabulary (same as used to train the model)
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab_ubuntu

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# Run coherence evaluation
[coherence]
type=pimlico.modules.gensim.coherence
input_topics_top_words=top_words
input_corpus=europarl
input_vocab=vocab
coherence=c_npmi
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.gensim.coherence*

dtm_train

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Train a DTM model on a pre-prepared word-ID corpus and document labels
#
# For a fuller example (on which this test is based), see
# :doc:`the topic model training example </example_config/topic_modelling.train_tms>`.
```

(continues on next page)

(continued from previous page)

```
[pipeline]
name=dtm_train
release=latest

# Load word IDs
[ids]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=IntegerListsDocumentType
dir=%(test_data_dir)s/datasets/corpora/ids_ubuntu

# Load slice labels
[labels]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=LabelDocumentType
dir=%(test_data_dir)s/datasets/corpora/labels_ubuntu

# Load vocabulary
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab_ubuntu

[dtm]
type=pimlico.modules.gensim.ldaseq
input_corpus=ids
input_labels=labels
input_vocab=vocab
# Small number of topics: you probably want more in practice
num_topics=2
# Speed up training for this test by reducing all passes/iterations to very small␣
↪values
em_min_iter=1
em_max_iter=1
passes=2
lda_inference_max_iter=2
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.gensim.ldaseq*

lda_top_words

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Extract lists of words from an LDA model
#
# These can be used for coherence evaluation.
```

(continues on next page)

(continued from previous page)

```
[pipeline]
name=lda_top_words
release=latest

# Load trained model
[lda]
type=pimlico.datatypes.gensim.GensimLdaModel
dir=%(test_data_dir)s/datasets/gensim/lda

# Extract the top words for each topic
[top_words]
type=pimlico.modules.gensim.lda_top_words
input_model=lda
num_words=10
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.gensim.lda_top_words*

dtm_infer

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Take a trained DTM model and perform inference on other docs
#
# For a fuller example (on which this test is based), see
# :doc:`the topic model training example </example_config/topic_modelling.train_tms>`.

[pipeline]
name=dtm_infer
release=latest

# Load word IDs
[ids]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=IntegerListsDocumentType
dir=%(test_data_dir)s/datasets/corpora/ids_ubuntu

# Load slice labels
[labels]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=LabelDocumentType
dir=%(test_data_dir)s/datasets/corpora/labels_ubuntu

# Load a trained DTM model
[dtm]
```

(continues on next page)

(continued from previous page)

```
type=pimlico.datatypes.gensim.GensimLdaSeqModel
dir=%(test_data_dir)s/datasets/dtm_model

# Apply stationary DTM inference to all of the documents
# This doesn't need to be run on the same document set we trained on:
#  we do that here just as an example
[dtm_infer]
type=pimlico.modules.gensim.ldaseq_doc_topics
input_corpus=ids
input_labels=labels
input_model=dtm
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.gensim.ldaseq_doc_topics*

opennlp_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=opennlp_tokenize
release=latest

# Prepared tarred corpus
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

# There's a problem with the tests here
# Pimlico still has a clunky old Makefile-based system for installing model data for␣
↪modules
# The tests don't know that this needs to be done before the pipeline can be run
# This is why this test is not in the main suite, but a special OpenNLP one
[tokenize]
type=pimlico.modules.opennlp.tokenize
token_model=en-token.bin
sentence_model=en-sent.bin
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.opennlp.tokenize*

opennlp_parse

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# The input data from Europarl has very long sentences, which makes the parser slow.
# It would be better to run the tests on input that would not take so long
[pipeline]
name=opennlp_parse
release=latest

# Prepared tarred corpus
[tokens]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# There's a problem with the tests here
# Pimlico still has a clunky old Makefile-based system for installing model data for␣
↪modules
# The tests don't know that this needs to be done before the pipeline can be run
# This is why this test is not in the main suite, but a special OpenNLP one
[parse]
type=pimlico.modules.opennlp.parse
model=en-parser-chunking.bin
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.opennlp.parse*

opennlp_pos

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=opennlp_pos
release=latest

# Prepared tarred corpus
[tokens]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized
```

(continues on next page)

(continued from previous page)

```
# There's a problem with the tests here
# Pimlico still has a clunky old Makefile-based system for installing model data for␣
↪modules
# The tests don't know that this needs to be done before the pipeline can be run
# This is why this test is not in the main suite, but a special OpenNLP one
[pos]
type=pimlico.modules.opennlp.pos
model=en-pos-maxent.bin
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.opennlp.pos*

embedding_norm

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Output trained embeddings in the word2vec format for external use
[pipeline]
name=embedding_norm
release=latest

# Take trained embeddings from a prepared Pimlico dataset
[embeddings]
type=pimlico.datatypes.embeddings.Embeddings
dir=%(test_data_dir)s/datasets/embeddings

# Apply L2 normalization: scale all vectors to unit length
[norm]
type=pimlico.modules.embeddings.normalize
l2_norm=T
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.normalize*

word2vec_train

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=word2vec_train
release=latest

# Take tokenized text input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[word2vec]
type=pimlico.modules.embeddings.word2vec
# Set low, since we're training on a tiny corpus
min_count=1
# Very small vectors: usually this will be more like 100 or 200
size=10
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.word2vec*

word2vec_store

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Output trained embeddings in the word2vec format for external use
[pipeline]
name=word2vec_store
release=latest

# Take trained embeddings from a prepared Pimlico dataset
[embeddings]
type=pimlico.datatypes.embeddings.Embeddings
dir=%(test_data_dir)s/datasets/embeddings

[store]
type=pimlico.modules.embeddings.store_word2vec
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.store_word2vec*

glove_train

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Train GloVe embeddings on a tiny corpus
[pipeline]
name=glove_train
release=latest

# Take tokenized text input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[glove]
type=pimlico.modules.embeddings.glove
# Set low, since we're training on a tiny corpus
min_count=1
# TODO Set more options
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.glove*

tsvvec_store

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Output trained embeddings in the TSV format for external use
[pipeline]
name=tsvvec_store
release=latest

# Take trained embeddings from a prepared Pimlico dataset
[embeddings]
type=pimlico.datatypes.embeddings.Embeddings
dir=%(test_data_dir)s/datasets/embeddings

[store]
type=pimlico.modules.embeddings.store_tsv
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.store_tsv*

fasttext_train

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Train fastText embeddings on a tiny corpus
[pipeline]
name=fasttext_train
release=latest

# Take tokenized text input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[fasttext]
type=pimlico.modules.embeddings.fasttext
# Set low, since we're training on a tiny corpus
min_count=1
# Very small vectors: usually this will be more like 100 or 200
dim=10
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.embeddings.fasttext*

interleave

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=interleave
release=latest

# Take input from some prepared Pimlico datasets
```

(continues on next page)

(continued from previous page)

```
[europarl1]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[europarl2]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl2


[interleave]
type=pimlico.modules.corpora.interleave
input_corpora=europarl1,europarl2

[output]
type=pimlico.modules.corpora.format
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.interleave*
- *pimlico.modules.corpora.format*

list_filter

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=list_filter
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[filename_list]
type=StringList
dir=%(test_data_dir)s/datasets/europarl_filename_list

# Use the filename list to filter the documents
# This should leave 3 documents (of original 5)
[europarl_filtered]
type=pimlico.modules.corpora.list_filter
```

(continues on next page)

(continued from previous page)

```
input_corpus=europarl
input_list=filename_list
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.list_filter*

vocab_unmapper

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=vocab_unmapper
release=latest

# Load the prepared vocabulary
#  (created by the vocab_builder test pipeline)
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab

# Load the prepared word IDs
[ids]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=IntegerListsDocumentType
dir=%(test_data_dir)s/datasets/corpora/ids

# Perform the mapping from IDs to words
[tokens]
type=pimlico.modules.corpora.vocab_unmapper
input_vocab=vocab
input_ids=ids
oov=OOV
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.vocab_unmapper*

shuffle

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=shuffle
release=latest

# Read in some Europarl raw files
# Instead of using the pre-prepared corpus stored in the pipeline-internal format,
#  we use an input reader here. This means it wouldn't be possible to use
#  the shuffle module type, so we're forced to use shuffle_linear.
# Another solution is to use a store module and then shuffle, which may be preferable
[europarl]
type=pimlico.modules.input.text.raw_text_files
files=%(test_data_dir)s/datasets/europarl_en_raw/*


[shuffle]
type=pimlico.modules.corpora.shuffle_linear
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.shuffle_linear*

subset

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=subset
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[subset]
type=pimlico.modules.corpora.subset
size=1
offset=2

[output]
type=pimlico.modules.corpora.format
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.subset*
- *pimlico.modules.corpora.format*

concat

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=concat
release=latest

# Take input from some prepared Pimlico datasets
[europarl1]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[europarl2]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl2


[concat]
type=pimlico.modules.corpora.concat
input_corpora=europarl1,europarl2

[output]
type=pimlico.modules.corpora.format
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.concat*
- *pimlico.modules.corpora.format*

subsample

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=subsample
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[subsample]
type=pimlico.modules.corpora.subsample
p=0.8
seed=1
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.subsample*

vocab_builder

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=vocab_builder
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[vocab]
type=pimlico.modules.corpora.vocab_builder
threshold=2
limit=500
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.vocab_builder*

group

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=group
release=latest

# Read in some Europarl raw files
[europarl]
type=pimlico.modules.input.text.raw_text_files
files=%(test_data_dir)s/datasets/europarl_en_raw/*
encoding=utf8

[group]
type=pimlico.modules.corpora.group

[output]
type=pimlico.modules.corpora.format
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.group*
- *pimlico.modules.corpora.format*

split

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=split
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[split]
type=pimlico.modules.corpora.split
set1_size=2
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.split*

vocab_mapper

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=vocab_mapper
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# Load the prepared vocabulary
#  (created by the vocab_builder test pipeline)
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab

# Perform the mapping from words to IDs
[ids]
type=pimlico.modules.corpora.vocab_mapper
input_vocab=vocab
input_text=europarl
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.vocab_mapper*

store

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=store
release=latest

# Read in some Europarl raw files
[europarl]
type=pimlico.modules.input.text.raw_text_files
files=%(test_data_dir)s/datasets/europarl_en_raw/*
encoding=utf8

# Group works as a filter module, so its output is not stored.
# This pipeline shows how you can store the output from such a
#  module for static use by later modules.
# In this exact case, you don't gain anything by doing that, since
# the grouping filter is fast, but sometimes it could be desirable
# with other filters
[group]
type=pimlico.modules.corpora.group

[store]
type=pimlico.modules.corpora.store
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.group*
- *pimlico.modules.corpora.store*

stats

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=stats
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[stats]
type=pimlico.modules.corpora.corpus_stats
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.corpus_stats*

filter_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Essentially the same as the simple_tokenize test pipeline,
# but uses the filter=T parameter on the tokenizer.
# This can be applied to any document map module, so this
# is intended as a test for that feature, rather than for
# simple_tokenize

[pipeline]
name=filter_tokenize
release=latest

[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
# This corpus is actually tokenized text, but we treat it as raw text and apply the␣
↪simple tokenizer
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# Tokenize as a filter: this module is not executable
[tokenize]
type=pimlico.modules.text.simple_tokenize
filter=T

# Then store the output
# You wouldn't really want to do this, as it's equivalent to not using
# the tokenizer as a filter! But we're testing the filter feature
[store]
type=pimlico.modules.corpora.store
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.store*

vocab_mapper

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=vocab_mapper
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized_longer

# Load the prepared vocabulary
#  (created by the vocab_builder test pipeline)
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab

# Perform the mapping from words to IDs
[ids]
type=pimlico.modules.corpora.vocab_mapper
input_vocab=vocab
input_text=europarl
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.vocab_mapper*

vocab_counter

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=vocab_counter
release=latest

# Load the prepared vocabulary
#  (created by the vocab_builder test pipeline)
[vocab]
type=pimlico.datatypes.dictionary.Dictionary
dir=%(test_data_dir)s/datasets/vocab

# Load the prepared token IDs
#  (created by the vocab_mapper test pipeline)
[ids]
```

(continues on next page)

(continued from previous page)

```
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=IntegerListsDocumentType
dir=%(test_data_dir)s/datasets/corpora/ids

# Count the frequency of each word in the corpus
[counts]
type=pimlico.modules.corpora.vocab_counter
input_corpus=ids
input_vocab=vocab
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.vocab_counter*

shuffle

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=shuffle
release=latest

# Take input from a prepared Pimlico dataset
# This works fine with shuffle, since it's already stored in the pipeline-internal␣
↪format
# However, it wouldn't work with an input reader, since
#  the interface doesn't provide random access to docs
# Then you'd need to use shuffle_linear
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[shuffle]
type=pimlico.modules.corpora.shuffle
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.shuffle*

tokenized_formatter

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Test the tokenized text formatter
[pipeline]
name=tokenized_formatter
release=latest

# Take input from a prepared tokenized dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# Format the tokenized data using the default formatter,
# which is declared for the tokenized datatype
[format]
type=pimlico.modules.corpora.format
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.format*

simple_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=simple_tokenize
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
# This corpus is actually tokenized text, but we treat it as raw text and apply the␣
↪simple tokenizer
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[tokenize]
type=pimlico.modules.text.simple_tokenize
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.simple_tokenize*

normalize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=normalize
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[norm]
type=pimlico.modules.text.normalize
case=lower
remove_empty=T
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.normalize*

normalize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=normalize
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[norm]
type=pimlico.modules.text.text_normalize
```

(continues on next page)

(continued from previous page)

```
case=lower
strip=T
blank_lines=T
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.text_normalize*

simple_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=simple_tokenize
release=latest

# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
# This corpus is actually tokenized text, but we treat it as raw text and apply the␣
↪char tokenizer
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

[tokenize]
type=pimlico.modules.text.char_tokenize
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.char_tokenize*

europarl

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=europarl
release=latest

# Read in some Europarl raw files, using the special Europarl reader
[europarl]
type=pimlico.modules.input.text.europarl
files=%(test_data_dir)s/datasets/europarl_en_raw/*

[store]
type=pimlico.modules.corpora.store
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.store*

huggingface_dataset

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=huggingface_dataset
release=latest

# Load an example dataset from Huggingface
[hf_dataset]
type=pimlico.modules.input.text.huggingface
dataset=glue
name=mrpc
split=train
doc_name=idx
columns=sentence1,sentence2
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.input.text.huggingface*

raw_text_files_test

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=raw_text_files_test
release=latest

# Read in some Europarl raw files
[europarl]
type=pimlico.modules.input.text.raw_text_files
files=%(test_data_dir)s/datasets/europarl_en_raw/*

[store]
type=pimlico.modules.corpora.store
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.store*

xml_test

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Test for the XML input module
#
# Read in raw text data from the Estonian Reference Corpus:
# https://www.cl.ut.ee/korpused/segakorpus/
# We have a tiny subset of the corpus here. It can be read using
# the standard XML input module.

[pipeline]
name=xml_test
release=latest

# Read in some XML files from Est Ref
[input]
type=pimlico.modules.input.xml
files=%(test_data_dir)s/datasets/est_ref/*.tei
document_node_type=text
```

glove_input_test

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=glove_input_test
release=latest

# Read in some vectors
[vectors]
type=pimlico.modules.input.embeddings.glove
path=%(test_data_dir)s/input_data/glove/glove.small.300d.txt
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.input.embeddings.glove*

fasttext_input_test

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=fasttext_input_test
release=latest

# Read in some vectors
[vectors]
type=pimlico.modules.input.embeddings.fasttext
path=%(test_data_dir)s/input_data/fasttext/wiki.en_top50.vec
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.input.embeddings.fasttext*

spacy_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=spacy_tokenize
release=latest

# Prepared tarred corpus
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[tokenize]
type=pimlico.modules.spacy.tokenize
model=en_core_web_sm
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.spacy.tokenize*

spacy_parse_text

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=spacy_parse_text
release=latest

# Prepared tarred corpus
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[extract_nps]
type=pimlico.modules.spacy.extract_nps
model=en_core_web_sm
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.spacy.extract_nps*

spacy_parse_text

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=spacy_parse_text
release=latest

# Prepared tarred corpus
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[tokenize]
type=pimlico.modules.spacy.parse_text
model=en_core_web_sm
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.spacy.parse_text*

collect_files

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=collect_files
release=latest

# Read in some named file datasets
# This could be, for example, the output of the stats module
[named_files1]
type=NamedFileCollection
filenames=text_file.txt
dir=%(test_data_dir)s/datasets/named_files1

[named_files2]
type=NamedFileCollection
filenames=data.bin,text_file.txt
dir=%(test_data_dir)s/datasets/named_files2

[collect]
type=pimlico.modules.utility.collect_files
input=named_files1,named_files2
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.utility.collect_files*

nltk_nist_tokenize

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
[pipeline]
name=nltk_nist_tokenize
release=latest

# Prepared grouped corpus of raw text data
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

# Tokenize the data using NLTK's simple NIST tokenizer
[tokenize_euro]
type=pimlico.modules.nltk.nist_tokenize

# Another tokenization, using the non_european option
[tokenize_non_euro]
type=pimlico.modules.nltk.nist_tokenize
input=europarl
non_european=T
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.nltk.nist_tokenize*
- *pimlico.modules.nltk.nist_tokenize*

malt_parse

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# The input data from Europarl has very long sentences, which makes the parser slow.
# It would be better to run the tests on input that would not take so long
[pipeline]
name=malt_parse
release=latest

# Load pre-tagged data
# This is in word-annotation format and was produced by the OpenNLP tagger test␣
↪pipeline
[pos]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=WordAnnotationsDocumentType(fields=word,pos)
dir=%(test_data_dir)s/datasets/corpora/pos

[parse]
type=pimlico.modules.malt
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.malt*

embeddings_plot

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Plot trained embeddings
[pipeline]
name=embeddings_plot
release=latest

# Take trained embeddings from a prepared Pimlico dataset
[embeddings]
type=pimlico.datatypes.embeddings.Embeddings
dir=%(test_data_dir)s/datasets/embeddings

[plot]
type=pimlico.modules.visualization.embeddings_plot
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.visualization.embeddings_plot*

bar_chart

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Plot several numbers on a bar chart
[pipeline]
name=bar_chart
release=latest

[a]
type=NumericResult
dir=%(test_data_dir)s/datasets/results/A

[b]
type=NumericResult
dir=%(test_data_dir)s/datasets/results/C

[c]
type=NumericResult
dir=%(test_data_dir)s/datasets/results/C

[plot]
type=pimlico.modules.visualization.bar_chart
input=a,b,c
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.visualization.bar_chart*

filter_map

This is one of the test pipelines included in Pimlico's repository. See *Module test pipelines* for more details.

Config file

The complete config file for this test pipeline:

```
# Pipeline designed to test the use of a document
# map module as a filter. It uses the text normalization
# module and will therefore fail if that module's
# test is also failing, but since the module is so
# simple, this is unlikely

[pipeline]
name=filter_map
release=latest
```

(continues on next page)

(continued from previous page)

```
# Take input from a prepared Pimlico dataset
[europarl]
type=pimlico.datatypes.corpora.GroupedCorpus
data_point_type=TokenizedDocumentType
dir=%(test_data_dir)s/datasets/corpora/tokenized

# Apply text normalization
# Unlike the test text/normalize.conf, we apply this
# as a filter, so its result is not stored, but computed
# on the fly and passed straight through to the
# next module
[norm_filter]
type=pimlico.modules.text.normalize
# Use the general filter option, which can be applied
# to any document map module
filter=T
case=lower

# Store the result of the previous, filter, module.
# This is a stupid thing to do, since we could have
# just not used the module as a filter and had the
# same effect, but we do it here to test the use
# of a module as a filter
[store]
type=pimlico.modules.corpora.store
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.store*

1.6.2 Input data

Pimlico also comes with all the data necessary to run the pipelines. They all use very small datasets, so that they don't take long to run and can be easily distributed.

Some of the datasets are raw data, of the sort you might find in a distributed corpus, and these are used to test input readers for that type of data. Most, however, are stored in one of Pimlico's datatype formats, exactly as they were output from some other module (most often from another test pipeline), so that they can be read in to test one module in isolation.

1.6.3 Usage examples

In addition to providing unit testing for core Pimlico modules, test pipelines also function as a source of examples of each module's usage. They are for that reason linked to from the module's documentation, so that example usages can be easily found where available.

1.6.4 Running

To run test pipelines, you can use the script `test_pipeline.sh` in Pimlico's bin directory, e.g.:

```
./test_pipeline.sh ../test/data/pipelines/corpora/concat.conf output
```

This will load a single test pipeline from the given config file and execute the module named `output`.

There are also some suites of tests, specified as CSV files giving a number of config files and module names to execute for each. To run the main suite of test pipelines for Pimlico's core modules, run:

```
./all_test_pipelines.sh
```

1.7 Example pipelines

Pimlico comes with a number of example pipelines to demonstrate how to use it.

A more extensive set of examples is also provided in the form of *test pipelines*, which give a small example of the usage of individual core modules and are used as unit tests for the modules.

1.7.1 Available pipelines

empty_test

This is an example Pimlico pipeline.

The complete config file for this example pipeline is below. Source file

A basic, empty pipeline. Includes no modules at all.

Used to test basic loading of pipelines in one of the unit tests.

Pipeline config

```
[pipeline]
name=empty_test
# Always test with the latest version
release=latest

[vars]
var_name=testing a variable
```

train_tms_example

This is an example Pimlico pipeline.

The complete config file for this example pipeline is below. Source file

An example pipeline that loads some textual data and trains topic models on it using Gensim.

See the `src/` subdirectory for the module's code.

Pipeline config

```
[pipeline]
name=train_tms_example
release=latest
# We need a path to Python code here, since we use a custom module type
python_path=src/

[vars]
# Here we define where the example input corpus can be found
corpus_path=%(pimlico_root)s/examples/data/input/ubuntu_dialogue/dialogues_small.json

# Read in the raw text from the JSON files
[input_text]
type=tm_example.modules.input.ubuntu_dialogue
path=%(corpus_path)s
# Just use a small number of documents so we can train fast
# You should use a much bigger corpus for a real model
limit=600

# Also read in a label for each document consisting of the year+month from
# the timestamp
[input_labels]
type=tm_example.modules.input.ubuntu_dialogue_months
path=%(corpus_path)s
limit=600

[store_labels]
type=pimlico.modules.corpora.store
input=input_labels

# Tokenize the text using a simple tokenizer from NLTK
[tokenize]
type=pimlico.modules.spacy.tokenize
input=input_text

# Apply simple text normalization
# In a real topic modelling application, you might want to do lemmatization
#  or other types of more sophisticated normalization here
[normalize]
type=pimlico.modules.text.normalize
case=lower
min_word_length=3
remove_empty=T
remove_only_punct=T

# Build a vocabulary from the words used in the corpus
# This is used to map words in the corpus to IDs
[vocab]
type=pimlico.modules.corpora.vocab_builder
input=normalize
# Only include words that occur at least 5 times
threshold=5

[ids]
type=pimlico.modules.corpora.vocab_mapper
input_vocab=vocab
```

(continues on next page)

(continued from previous page)

```
input_text=normalize
# Skip any OOV words (below the threshold)
oov=ignore

# First train a plain LDA model using Gensim
[lda]
type=pimlico.modules.gensim.lda
input_vocab=vocab
input_corpus=ids
tfidf=T
# Small number of topics: you probably want more in practice
num_topics=5
passes=10

# Also train a dynamic topic model (DTM), with a separate model
# for each month
[dtm]
type=pimlico.modules.gensim.ldaseq
input_corpus=ids
input_labels=input_labels
input_vocab=vocab
# Small number of topics: you probably want more in practice
num_topics=5
# Apply TF-IDF transformation to bags of words before training
tfidf=T
# Speed up training for this demo by reducing iterations
em_min_iter=3
em_max_iter=8

# Apply stationary DTM inference to all of the documents
# This doesn't need to be run on the same document set we trained on:
#  we do that here just as an example
[dtm_infer]
type=pimlico.modules.gensim.ldaseq_doc_topics
input_corpus=ids
input_labels=input_labels
input_model=dtm
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.corpora.store*
- *pimlico.modules.spacy.tokenize*
- *pimlico.modules.text.normalize*
- *pimlico.modules.corpora.vocab_builder*
- *pimlico.modules.corpora.vocab_mapper*
- *pimlico.modules.gensim.lda*
- *pimlico.modules.gensim.ldaseq*
- *pimlico.modules.gensim.ldaseq_doc_topics*

custom_module_example

This is an example Pimlico pipeline.

The complete config file for this example pipeline is below. Source file

A simple example pipeline that loads some textual data and tokenizes it, then performs some custom processing.

This is intended as an example of how to use your own code in a pipeline module. The pipeline contains some core modules, but also one the is defined especially for this pipeline: `filter_prop_nns`.

See the `src/` subdirectory for the module's code.

Pipeline config

```
# Options for the whole pipeline
[pipeline]
name=custom_module_example
# Specify the version of Pimlico this config is designed to work with
release=latest
# Here you can add path(s) to Python source directories the pipeline needs
# We need to do this here, since we use a custom module type
# The path is relative to this config file
python_path=src/

# Specify some things in variables at the top of the file, so they're easy to find
[vars]
# The main pipeline input dir is given here
# It's good to put all paths to input data here, so that it's easy for people to point
#  them to other locations
# Here we define where the example input text data can be found
text_path=%(pimlico_root)s/examples/data/input/bbc/data

# Read in the raw text files
[input_text]
type=pimlico.modules.input.text.raw_text_files
files=%(text_path)s/*

# Tokenize the text using a simple tokenizer from NLTK
[tokenize]
type=pimlico.modules.spacy.tokenize
input=input_text

# A rough simple filter to remove words that look like proper nouns
# This is here to demonstrate how to use a custom module that is not
#  part of Pimlico's core modules
[filter_prop_nns]
type=pim_example.modules.filter_prop_nns
input=tokenize

# Build a vocabulary from the words used in the resulting corpus
# This can later be used to map words in the corpus to IDs
[vocab]
type=pimlico.modules.corpora.vocab_builder
input=filter_prop_nns
# Only include words that occur at least 5 times
threshold=5
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.spacy.tokenize*
- *pimlico.modules.corpora.vocab_builder*

tokenize_example

This is an example Pimlico pipeline.

The complete config file for this example pipeline is below. Source file

A simple example pipeline that loads some textual data and tokenizes it, using an extremely simple tokenizer.

This is an example of a simple pipeline, but not a good example of how to do tokenization. For real applications, you should use a proper, language-specific tokenizer, like the *OpenNLP one*, or at least *NLTK's NIST tokenizer*.

Pipeline config

```
# Options for the whole pipeline
[pipeline]
name=tokenize_example
# Specify the version of Pimlico this config is designed to work with
# It will run with any release that's the same major version and the same or a later␣
↪minor version
# Here we use "latest" so we're always running the example against the latest version,
#  but you should specify the version you wrote the pipeline for
release=latest
# Here you can add path(s) to Python source directories the pipeline needs
# Typically, you'll just add a single path, specified relative to the config file's␣
↪location
python_path=

# Specify some things in variables at the top of the file, so they're easy to find
[vars]
# The main pipeline input dir is given here
# It's good to put all paths to input data here, so that it's easy for people to point
#  them to other locations
# Here we define where the example input text data can be found
text_path=%(pimlico_root)s/examples/data/input/bbc/data

# Read in the raw text files
[input_text]
type=pimlico.modules.input.text.raw_text_files
files=%(text_path)s/*

# Tokenize the text using a very simple tokenizer
# For real applications, you should use a proper, language-specific tokenizer,
#  like the OpenNLP one, or at least NLTK's NIST tokenizer
[tokenize]
type=pimlico.modules.text.simple_tokenize
input=input_text
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.simple_tokenize*

tokenize_example2

This is an example Pimlico pipeline.

The complete config file for this example pipeline is below. Source file

A simple example pipeline that loads some textual data and tokenizes it, using an extremely simple tokenizer.

This is an example of a simple pipeline, but not a good example of how to do tokenization. For real applications, you should use a proper, language-specific tokenizer, like the *OpenNLP one*, or at least *NLTK's NIST tokenizer*.

Pipeline config

```
# Options for the whole pipeline
[pipeline]
name=tokenize_example2
# Specify the version of Pimlico this config is designed to work with
# It will run with any release that's the same major version and the same or a later␣
↪minor version
# Here we use "latest" so we're always running the example against the latest version,
#  but you should specify the version you wrote the pipeline for
release=latest
# We need to load the input reader type, which is the same one used for the topic
# modelling example
python_path=../topic_modelling/src/

# Specify some things in variables at the top of the file, so they're easy to find
[vars]
# The main pipeline input dir is given here
# It's good to put all paths to input data here, so that it's easy for people to point
#  them to other locations
# Here we define where the example input text data can be found
text_path=%(pimlico_root)s/examples/data/input/ubuntu_dialogue/dialogues_bigger.json

# Read in the raw text from the JSON files
[input_text]
type=tm_example.modules.input.ubuntu_dialogue
path=%(text_path)s

# Tokenize the text using a very simple tokenizer
# For real applications, you should use a proper, language-specific tokenizer,
#  like the OpenNLP one, or at least NLTK's NIST tokenizer
[tokenize]
type=pimlico.modules.text.simple_tokenize
input=input_text
```

Modules

The following Pimlico module types are used in this pipeline:

- *pimlico.modules.text.simple_tokenize*

1.7.2 Running

To run example pipelines, you can use the script `run_example.sh` in Pimlico's `example` directory, e.g.:

```
./example_pipeline.sh simple/tokenize.conf status
```

This will load a single example pipeline from the given config file and show the execution status of the modules.

1.8 Future plans

Development of Pimlico is constantly ongoing. A lot of this involves adding new *core module types*. There are also planned feature enhancements.

1.8.1 Wishlist

Things I plan to add to Pimlico.

- *Pipeline graph visualizations*. Maybe an interactive GUI to help with viewing large pipelines
- Model fetching: system like software dependency checking and installation to download models on demand
- See issue list on Github for other specific plans

Module types to be updated, implemented in the old datatypes system (using backwards incompatible library features). These do not take long to update and include in the main library.

- C&C parser
- CoreNLP tools (switch to using Stanza wrappers, see below)
- Compiling term-feature matrices (for count-based embeddings among other things)
- Building count-based embeddings from dependency features
- OpenNLP tools. Some already updated. To do:
  - Coreference resolution
  - Coreference pipeline (from raw text)
  - NER
- R-script: simple module to run an arbitrary R script
- Scikit-learn matrix factorization. (Lots of Scikit-learn modules could be added, but this one already exists in an old form.)
- Copy file utility: output a file to a given location outside the pipeline-internal storage
- Bar chart visualization

New module types

- Stanza tools: Stanford's new toolkit, includes Python bindings and CoreNLP wrappers
- More spaCy tools: currently only have tokenizer

More details on some of these plans

Berkeley Parser

https://github.com/slavpetrov/berkeleyparser

Java constituency parser. Pre-trained models are also provided in the Github repo.

Probably no need for a Java wrapper here. The parser itself accepts input on stdin and outputs to stdout, so just use a subprocess with pipes.

Cherry Picker

Coreference resolver

http://www.hlt.utdallas.edu/~altaf/cherrypicker/

Requires NER, POS tagging and constituency parsing to be done first. Tools for all of these are included in the Cherry Picker codebase, but we just need a wrapper around the Cherry Picker tool itself to be able to feed these annotations in from other modules and perform coref.

Write a Java wrapper and interface with it using Py4J, as with OpenNLP.

Outputting pipeline diagrams

Once pipeline config files get big, it can be difficult to follow what's going on in them, especially if the structure is more complex than just a linear pipeline. A useful feature would be the ability to display/output a visualization of the pipeline as a flow graph.

It looks like the easiest way to do this will be to construct a DOT graph using Graphviz/Pydot and then output the diagram using Graphviz.

http://www.graphviz.org

https://pypi.python.org/pypi/pydot

Building the graph should be pretty straightforward, since the mapping from modules to nodes is fairly direct.

We could also add extra information to the nodes, like current execution status.

1.8.2 Todos

The following to-dos appear elsewhere in the docs. They are generally bits of the documentation I've not written yet, but am aware are needed.

Todo: This has not been updated for the Pimarc internal storage format, so still assumes that tar files are used. It will be updated in future, if there is a need for it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/src/python/pimlico/cli/recov
of pimlico.cli.recover.RecoverCmd, line 4.)

Todo: In future, this should be replaced by a doc type that reads in the parse trees and returns a tree data structure. For now, you need to load and process the tree strings yourself.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/src/python/pimlico/datatype
of pimlico.datatypes.corpora.parse.trees.OpenNLPTreeStringsDocumentType, line 4.)

Todo: Add unit test for ScoredReadFeatureSets

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/src/python/pimlico/datatype of pimlico.datatypes.features.ScoredRealFeatureSets, line 9.)

Todo: Not got these things working yet, but they'll be useful in the long run

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/src/python/pimlico/utils/urv of pimlico.utils.urwid, line 8.)

Todo: This has not been updated for the Pimarc internal storage format, so still assumes that tar files are used. It will be updated in future, if there is a need for it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/commands/recover.rst, line 13.)

Todo: Describe how module dependencies are defined for different types of deps

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/dependencies.rst, line 73.)

Todo: Include some examples from the core modules of how deps are defined and some special cases of software fetching

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/dependencies.rst, line 80.)

Todo: Finish the missing parts of this doc below

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 9.)

Todo: Document optional outputs.

Should include choose_optional_outputs_from_options(options, inputs) for deciding what optional outputs to include.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 170.)

Todo: Fully document module options, including: required, type checking/processing and other fancy features.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 221.)

Todo: Further document specification of software dependencies

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 239.)

Todo: This section is copied from *Pimlico module structure*. It needs to be re-written to provide more technical and comprehensive documentation of module execution.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 249.)

Todo: This section is copied from *Pimlico module structure*. It needs to be re-written to provide more technical and comprehensive documentation of pipeline config. NB: config files are fully documented in *Pipeline config*, so this just covers how ModuleInfo relates to the config.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/core/module_structure line 313.)

Todo: Filter module guide needs to be updated for new datatypes. This section is currently completely wrong – **ignore it!** This is quite a substantial change.

The difficulty of describing what you need to do here suggests we might want to provide some utilities to make this easier!

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/guides/filters.rst, line 31.)

Todo: Write a guide to building document map modules.

For now, the skeletons below are a useful starting point, but there should be a more fulsome explanation here of what document map modules are all about and how to use them.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/guides/map_module.rs line 5.)

Todo: Document map module guides needs to be updated for new datatypes.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/guides/map_module.rs line 12.)

Todo: Module writing guide needs to be updated for new datatypes.

In particular, the executor example and datatypes in the module definition need to be updated.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/guides/module.rst, line 23.)

Todo: Currently, this accepts any GroupedCorpus as input, but checks at runtime that the input is stored used the pipeline-internal format. It would be much better if this check could be enforced at the level of datatypes, so that the input datatype requirement explicitly rules out grouped corpora coming from input readers, filters or other dynamic sources.

Since this requires some tricky changes to the datatype system, I'm not implementing it now, but it should be done in future.

It will be implemented as part of the replacement of `GroupedCorpus` by `StoredIterableCorpus`: **'https://github.com/markgw/pimlico/issues/24'_**

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/modules/pimlico.modu line 27.)

Todo: Add test pipeline and test

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/modules/pimlico.modu line 21.)

Todo: Add test pipeline. This is slightly difficult, as we need a small FastText binary file, which is harder to produce, since you can't easily just truncate a big file.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/modules/pimlico.modu line 29.)

Todo: Add test pipeline

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/modules/pimlico.modu line 15.)

Todo: Add test pipeline

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/latest/docs/modules/pimlico.modu line 47.)

Note: New datatypes system

Some time ago, we changed how datatypes work internally, requiring all datatypes and modules to be updated. More info...

This has been done for many of the core modules, but some are waiting to be updated and don't work. They do not appear in the documentation and can be found in `pimlico.old_datatypes.modules`.

These issues will be resolved before v1.0 is released.

- genindex
- search

Python Module Index

c

d

m

p

t

u

Index

A

B

C

D

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y