Pimlico Documentation

Release 0.9.23

Mark Granroth-Wilding

Aug 07, 2019

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.

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.

Note: These are the docs for the **release candidate for v1.0**.

This brings with it a big project to change how datatypes work internally (previously in branch `datatypes`) and requires all datatypes and modules to be updated to the new system. More info...

Modules marked with `!!` in the docs are waiting to be updated and don't work. Other known outstanding tasks are marked with todos: *full todo list*.

These issues will be resolved before v1.0 is released.

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

- to provide **clear documentation** of what has been done;
- to make it easy to **incorporate standard NLP tasks**,
- and to extend the code with **non-standard tasks, specific to a pipeline**;
- to support simple **distribution of code** for reproduction, for example, on other datasets.

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. To get started with a Pimlico project, follow the *getting-started guide*.

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

More NLP tools will gradually be added. See *my wishlist* for current plans.

CHAPTER 1

Contents

1.1 Pimlico guides

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

1.1.1 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
```

1.1.2 Setting up a new project using Pimlico

Todo: Setup guide has a lot that needs to be updated for the new datatypes system. I've updated up to **Getting input**.

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

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.

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 major version (the first digit). Otherwise, 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/*
```

Todo: Continue writing from here

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. The OpenNLP-based tokenization module does both of these things at once, calling OpenNLP tools.

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.opennlp.tokenize
input=tar-grouper
```

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

Before we can run it, there's one thing missing: three of these modules have their own dependencies, so we need to get hold of the libraries they use. The input reader uses the Beautiful Soup python library and the tokenization and POS tagging modules use OpenNLP.

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
```

With any luck, all the checks will be successful. There might be some missing software dependencies.

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.

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
```

In this case, it will tell you that some libraries are missing, but they can be installed automatically. Simply issue the `install` command for the module.

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

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'll notice that the `tar-grouper` module doesn't feature in the list. This is because it's a filter – it's run on the fly while reading output from the previous module (i.e. the input), so doesn't have anything to run itself.

You might be surprised to see that `input-text` *does* feature 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 modules*
- *Writing document map modules*
- *Pimlico module structure*

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

1.1.3 Running a 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.

1.1.4 Writing Pimlico modules

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/
        ...
    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 example, 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,
```

(continues on next page)

(continued from previous page)

```
        },
    }

    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
```

1.1.5 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,
```

(continues on next page)

(continued from previous page)

```
        },
    }

    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


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):
```

(continues on next page)

(continued from previous page)

```
    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,
)
```

1.1.6 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.

1.1.7 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
            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
    # 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,
                [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
↪")],
                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.

1.1.8 Running one pipeline 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.

1.1.9 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.

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 special type 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
  ```

(continues on next page)

(continued from previous page)

```
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.

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

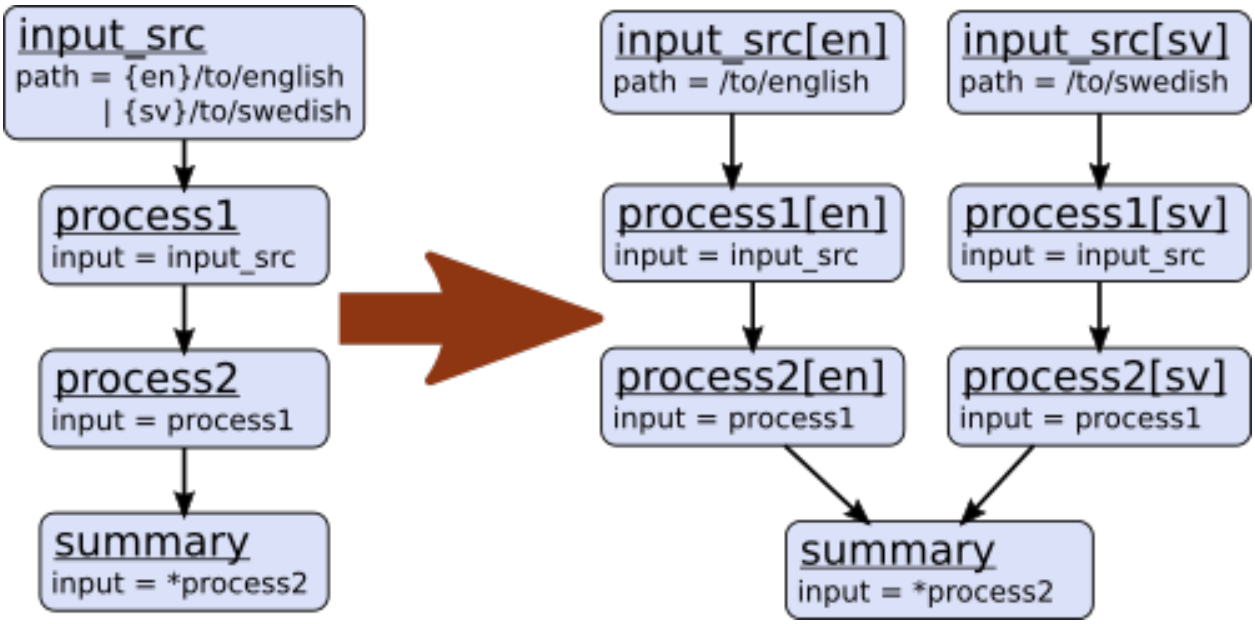
Module variables are set by including parameters in a module's config of the form `modvar_<name> = <value>`. This will assign `value` to the variable `name` for this module. The simplest form of assignment is just a string literal, enclosed in double quotes:

```
[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 modules*.

Todo: Write documentation for this

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`.

---

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

Defining module dependencies

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

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.

1.3.1 !! candc

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.2 !! corenlp

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.3 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.

- *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.

- *store*
- *group*

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.

It is difficult to do this efficiently for a large corpus. 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.

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
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
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`

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
```

Test pipelines

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

- *filter_map*
- *filter_tokenize*

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.

Inputs

Name	Type(s)
text	grouped_corpus <TokenizedDocumentType>

Outputs

Name	Type(s)
vocab	dictionary

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_proportion	Include terms that occur in max this proportion of documents	float
oov	Use the final index the represent chars that will be out of vocabulary after applying threshold/limit filters. Applied even if the count is 0. Represent OOVs using the given string in the vocabulary	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=value
prune_at=2000000
threshold=100
```

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

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
```

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*

1.3.4 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.

!! dependencies

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

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.

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.

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*

1.3.5 Feature set processing

Various tools for generic processing of extracted sets of features: building vocabularies, mapping to integer indices, etc.

!! term_feature_compiler

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! term_feature_matrix_builder

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! vocab_builder

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! vocab_mapper

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.6 Gensim topic modelling

Modules providing access to topic model training and other routines from Gensim.

LDA trainer

Path	pimlico.modules.gensim.lda
Executable	yes

Trains LDA using Gensim's basic LDA implementation, or the multicore version.

Todo: Add test pipeline and test

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
```

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.

Todo: Add test pipeline and test

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
```

1.3.7 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

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.

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_embedding_reader_module]
type=pimlico.modules.input.embeddings.fasttext
path=value
```

This example usage includes more options.

```
[my_fasttext_embedding_reader_module]
type=pimlico.modules.input.embeddings.fasttext
limit=0
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`).

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.

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
```

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.

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.

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	`embeddings`

Options

Name	Description	Type
binary	Assume input is in word2vec binary format. Default: True	bool
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
path=value
```

Text corpora

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
```

(continues on next page)

(continued from previous page)

```
encoding_errors=strict
exclude=path1,path2,...
files=path1,path2,...
```

Annotated text

Datasets that store text with accompanying annotations, like POS tags or dependency parses.

There are lots of different ways of storing this type of data in common usage. Here we currently only implement variants on one – the VRT format, used by Korp. In future, others should be added, e.g. CoNLL dependency parses.

Datatypes exist for some of these, which should be converted to input readers in due course.

VRT annotated text files

Path	pimlico.modules.input.text_annotations.vrt_text
Executable	yes

Input reader for VRT text collections (VeRticalized Text, as used by Korp:), just for reading the (tokenized) text content, throwing away all the annotations.

Uses sentence tags to divide each text into sentences.

See also:

`pimlico.modules.input.text_annotations.vrt`: Reading VRT files with all their annotations

Todo: Update to new datatypes system and add test pipeline

Todo: Currently skipped from module doc generator, until updated

---

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 <TokenizedDocumentType>`

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_vrt_files_reader_module]
type=pimlico.modules.input.text_annotations.vrt_text
files=path1,path2,...
```

This example usage includes more options.

```
[my_vrt_files_reader_module]
type=pimlico.modules.input.text_annotations.vrt_text
archive_basename=archive
archive_size=1000
encoding=utf8
encoding_errors=strict
exclude=path1,path2,...
files=path1,path2,...
```

Raw text 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_raw_text_files_reader_module]
type=pimlico.modules.input.xml
files=path1,path2,...
```

This example usage includes more options.

```
[my_raw_text_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
```

1.3.8 Malt dependency parser

Wrapper around the Malt dependency parser and data format converters to support connections to other modules.

!! conll_parser_input

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! parse

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.9 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*

1.3.10 OpenNLP modules

A collection of module types to wrap individual OpenNLP tools.

!! coreference

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! coreference_pipeline

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! ner

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! parse

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! pos

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

OpenNLP 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*

1.3.11 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
```

1.3.12 R interfaces

Modules for interfacing with the statistical programming language R. Currently, we provide just a simple way to pass data from the output of another module into an R script and run it. In the future, it may be appropriate to add more sophisticated interfaces, or expose R's functionality in a more specialised way, integrating more closely with Pimlico's datatype system.

!! script

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.13 Regular expressions

!! annotated_text

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.14 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"
```

!! matrix_factorization

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.15 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 only the following:

- case normalization (to upper or lower case)
- blank line removal
- empty sentence removal

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 ''
remove_empty	Skip over any empty sentences (i.e. blank lines)	bool
remove_only_punct	Skip over any sentences that are empty if punctuation is ignored	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=
remove_empty=F
remove_only_punct=F
```

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=
```

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*

1.3.16 General utilities

General utilities for things like filesystem manipulation.

!! alias

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! collect_files

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

!! copy_file

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

1.3.17 Visualization tools

Modules for plotting and suchlike

!! bar_chart

Note: This module has not yet been updated to the new datatype system, so cannot be used yet. Soon it will be updated.

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
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
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] [--short] [--history] [--deps-of␣
↪DEPS_OF] [--no-color]
```

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
`--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

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}]
```

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`, `-e`	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

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.

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]
```

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

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 one pipeline 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 one pipeline 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]
```

1.5 API Documentation

API documentation for the main Pimlico codebase, excluding the *built-in Pimlico module types*.

1.5.1 pimlico package

Subpackages

pimlico.cli package

Subpackages

pimlico.cli.browser package

Subpackages

pimlico.cli.browser.tools package

Submodules

pimlico.cli.browser.tools.corpus module

pimlico.cli.browser.tools.files module

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

pimlico.cli.browser.tools.formatter module

Module contents

Submodules

pimlico.cli.browser.tool module

Tool for browsing datasets, reading from the data output by pipeline modules.

browse_cmd(*pipeline*, *opts*)
Command for main Pimlico CLI

Module contents

pimlico.cli.debug package

Submodules

pimlico.cli.debug.stepper module

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*)

pimlico.cli.shell package

Submodules

pimlico.cli.shell.base module

class ShellCommand
Bases: `future.types.newobject.newobject`

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()

postloop()

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*)

exception ShellError
Bases: exceptions.Exception

pimlico.cli.shell.commands module

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

pimlico.cli.shell.runner module

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

pimlico.cli.check module

pimlico.cli.clean module

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*)

pimlico.cli.fixlength module

pimlico.cli.loaddump module

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 one pipeline 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 one pipeline 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 du

command_desc = "Load a module's output data from a tarball previously created by the du

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

pimlico.cli.locations module

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 datase

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*)

pimlico.cli.main module

pimlico.cli.newmodule module

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*)

pimlico.cli.pyshell module

class PimlicoPythonShellContext
Bases: `future.types.newobject.newobject`

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: `exceptions.Exception`

pimlico.cli.recover module

pimlico.cli.reset module

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*)

pimlico.cli.run module

pimlico.cli.status module

class StatusCmd
Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'status'

command_help = 'Output a module execution schedule for the pipeline and execution stat

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

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.

module_status(*module*)
Detailed module status, shown when a specific module's status is requested.

pimlico.cli.subcommands module

class PimlicoCLISubcommand

Bases: `future.types.newobject.newobject`

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*)

pimlico.cli.testemail module

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*)

pimlico.cli.util module

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

pimlico.core package

Subpackages

pimlico.core.dependencies package

Submodules

pimlico.core.dependencies.base module

Base classes for defining software dependencies for module types and routines for fetching them.

class SoftwareDependency(*name*, *url=None*, *dependencies=None*)

Bases: `future.types.newobject.newobject`

Base class for all Pimlico module software dependencies.

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 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*)
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: exceptions.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.

pimlico.core.dependencies.core module

Basic Pimlico core dependencies

CORE_PIMLICO_DEPENDENCIES = [PythonPackageSystemwideInstall<Pip>, PythonPackageOnPip<Pip (p
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.

pimlico.core.dependencies.java module

pimlico.core.dependencies.python module

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*, ***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.

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*, *url=None*, *dependencies=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).

pimlico.core.dependencies.versions module

class SoftwareVersion(*string_id*)
Bases: `future.types.newobject.newobject`

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

pimlico.core.external package

Submodules

pimlico.core.external.java module

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: `future.types.newobject.newobject`

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: `exceptions.Exception`

exception DependencyCheckerError
Bases: `exceptions.Exception`

exception JavaProcessError
Bases: `exceptions.Exception`

Module contents

Tools for calling external (non-Python) tools.

pimlico.core.modules package

Subpackages

pimlico.core.modules.map package

Submodules

pimlico.core.modules.map.filter module

pimlico.core.modules.map.multiproc module

pimlico.core.modules.map.singleproc module

pimlico.core.modules.map.threaded module

Module contents

Submodules

pimlico.core.modules.base module

pimlico.core.modules.execute module

pimlico.core.modules.inputs module

pimlico.core.modules.multistage module

pimlico.core.modules.options module

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 'future.types.newstr.newstr'>*, *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: exceptions.Exception

Module contents

Core functionality for loading and executing different types of pipeline module.

Submodules

pimlico.core.config module

pimlico.core.logs module

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

pimlico.core.paths module

abs_path_or_model_dir_path (*path*, *model_type*)

Module contents

pimlico.datatypes package

Subpackages

pimlico.datatypes.corpora package

Submodules

Submodules

pimlico.test.pipeline module

pimlico.test.suite module

Module contents

pimlico.utils package

Subpackages

pimlico.utils.docs package

Submodules

pimlico.utils.docs.commandgen module

pimlico.utils.docs.modulegen module

pimlico.utils.docs.rest module

make_table(*grid*, *header=None*)

table_div(*col_widths*, *header_flag=False*)

normalize_cell(*string*, *length*)

pimlico.utils.docs.testgen module

Module contents

trim_docstring(*docstring*)

Submodules

pimlico.utils.communicate module

timeout_process(**args*, ***kwds*)

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: future.types.newobject.newobject

length

encode()

static read(*stream*)

exception StreamCommunicationError
Bases: exceptions.Exception

pimlico.utils.core module

multiwith(**args*, ***kwds*)
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: `future.types.newobject.newobject`

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

pimlico.utils.email module

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: `future.types.newobject.newobject`

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: exceptions.Exception

pimlico.utils.filesystem module

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).

pimlico.utils.format module

multiline_tablate(*table*, *widths*, ***kwargs*)

title_box(*title_text*)
Make a nice big pretty title surrounded by a box.

pimlico.utils.linguistic module

strip_punctuation(*s*, *split_words=True*)

pimlico.utils.logging module

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

pimlico.utils.network module

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.

pimlico.utils.pipes module

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: `future.types.newobject.newobject`

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.

pimlico.utils.pos module

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: `exceptions.Exception`

pimlico.utils.probability module

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.

pimlico.utils.progress module

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.

update(*value=None*)
Updates the ProgressBar to a new value.

increment()

class DummyFileDescriptor
Bases: `future.types.newobject.newobject`

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: `future.types.newobject.newobject`

pimlico.utils.strings module

truncate(*s*, *length*, *ellipsis=u'...'*)

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.

pimlico.utils.system module

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

pimlico.utils.timeout module

timeout(*func*, *args=()*, *kwargs={}*, *timeout_duration=1*, *default=None*)

pimlico.utils.urwid module

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: exceptions.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*)

pimlico.utils.web module

download_file(*url*, *target_file*)
Now just an alias for urllib.urlretrieve()

Module contents

Submodules

pimlico.cfg module

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()

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

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:

- *nist_tokenize*
- *nist_tokenize*

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:

- *char_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
```

(continues on next page)

(continued from previous page)

```
data_point_type=RawTextDocumentType
dir=%(test_data_dir)s/datasets/text_corpora/europarl

[norm]
type=pimlico.modules.text.text_normalize
case=lower
strip=T
blank_lines=T
```

Modules

The following Pimlico module types are used in this pipeline:

- *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=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:

- *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␣
↪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:

- *simple_tokenize*

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:

- *embeddings_plot*

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

# 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:

- *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
```

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/*
```

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
```

(continues on next page)

(continued from previous page)

```
[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:

- *fasttext*

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:

- *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]
```

(continues on next page)

(continued from previous page)

```
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:

- *store_tsv*

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:

- *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:

- *store_word2vec*

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:

- *tokenize*

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:

- *group*
- *store*

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
[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:

- *shuffle*

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:

- *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

# 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:

- *vocab_mapper*

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:

- *concat*
- *format*

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:

- *group*
- *format*

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:

- *vocab_builder*

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
```

(continues on next page)

(continued from previous page)

```
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:

- *subset*
- *format*

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
[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:

- *interleave*
- *format*

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]
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:

- *vocab_counter*

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:

- *corpus_stats*

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
input_corpus=europarl
input_list=filename_list
```

Modules

The following Pimlico module types are used in this pipeline:

- *list_filter*

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:

- *split*

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:

- *format*

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 Future plans

Various things I plan to add to Pimlico in the futures. For a summary, see *Pimlico Wishlist*.

1.7.1 Pimlico Wishlist

Things I plan to add to Pimlico.

- Further modules:
  - *CherryPicker* for coreference resolution
  - *Berkeley Parser* for fast constituency parsing
  - Reconcile coref. Seems to incorporate upstream NLP tasks. Would want to interface such that we can reuse output from other modules and just do coref.

- **Pipeline graph visualizations**: *Outputting pipeline diagrams*. Maybe an interactive GUI to help with viewing large pipelines
- See issue list on Github for other specific plans
- Big redesign of datatype implementation is documented as a Github project

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: 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/python3/src/python/pimlico/utils/
of pimlico.utils.urwid, line 8.)

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/python3/docs/core/dependencies.r
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/python3/docs/core/dependencies.r
line 80.)

Todo: Write documentation for this

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/core/module_struct
line 9.)

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/python3/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/python3/docs/guides/map_modul
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/python3/docs/guides/map_modul 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/python3/docs/guides/module.rst, line 23.)

Todo: Setup guide has a lot that needs to be updated for the new datatypes system. I've updated up to **Getting input**.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/guides/setup.rst, line 5.)

Todo: Continue writing from here

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/guides/setup.rst, line 110.)

Todo: Add test pipeline and test

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/modules/pimlico.m line 15.)

Todo: Add test pipeline and test

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/modules/pimlico.m line 19.)

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/python3/docs/modules/pimlico.m line 27.)

Todo: Update to new datatypes system and add test pipeline

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/modules/pimlico.m line 24.)

Todo: Currently skipped from module doc generator, until updated

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/modules/pimlico.m
line 28.)

Todo: Add test pipeline

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/python3/docs/modules/pimlico.m
line 15.)

1.7.2 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.

1.7.3 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.

1.7.4 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.

- genindex
- search

Python Module Index

c

m

p

t

u

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Y