Pimlico Documentation

Release 0.8

Mark Granroth-Wilding

May 31, 2018

Contents

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

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

```
long_term_store=/home/me/.pimlico_store
short_term_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

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.

System-wide configuration

Pimlico needs you to specify certain parameters regarding your local system. In the simplest case, this is just a file in your home directory called `.pimlico`. See *Local configuration* for more details.

It needs to know where to put output files as it executes. Settings are given in a config file in your home directory and apply to all Pimlico pipelines you run. Note that Pimlico will make sure that different pipelines don't interfere with each other's output (provided you give them different names).

There are two locations you need to specify: **short-term** and **long-term storage**. For more details, see *Long-term and short-term stores*.

For a simple setup, these could be just two subdirectories of the same directory. However, it can be useful to distinguish them.

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

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

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

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/` directory) and creates a basic config file template, which will define your pipeline.

It also retrieves some 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 &ndash; 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 XML files and iterates of `<doc>` tags to get documents. This is how the Gigaword corpus is stored, so if you have Gigaword, just set the path to point to it.

Todo: Use a dataset that everyone can get to in the example

```
[input-text]
type=pimlico.datatypes.XmlDocumentIterator
path=/path/to/data/dir
```

Perhaps your corpus is very large and you'd rather try out your pipeline on a small subset. In that case, add the following option:

```
truncate=1000
```

Note: For a neat way to define a small test version of your pipeline and keep its output separate from the main pipeline, see *Pipeline variants*.

Grouping files

The standard approach to storing data between modules in Pimlico is to group them together into batches of documents, storing each batch in a tar archive, containing a file for every document. This works nicely with large corpora, where having every document as a separate file would cause filesystem difficulties and having all documents in the same file would result in a frustratingly large file.

We can do the grouping on the fly as we read data from the input corpus. The `tar_filter` module groups documents together and subsequent modules will all use the same grouping to store their output, making it easy to align the datasets they produce.

```
[tar-grouper]
type=pimlico.modules.corpora.tar_filter
input=input-text
```

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 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.

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,
        },
    }
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
        nmf = NMF(components)

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

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

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

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

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

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

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

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

Pipeline config

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

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

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

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

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

Skeleton new module

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

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

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

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

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

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

        # DO STUFF

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

1.1.4 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.

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"
```

(continues on next page)

(continued from previous page)

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

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

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

A bare-bones executor:

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


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

    # Return an object to send to the writer
    return output


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):
```

(continues on next page)

(continued from previous page)

```
    pass


def set_up_worker(worker):
    pass


def tear_down_worker(worker, error=None):
    pass


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

1.1.5 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

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.

1.1.6 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.

1.1.7 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.

1.1.8 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).
- **`long_term_store`:** Long-term store base directory being used under the current config. Can be used to link to data from other pipelines run on the same system. This is the value specified in the *local config file*.
- **`short_term_store`:** Short-term store base directory being used under the current config. Can be used to link to data from other pipelines run on the same system. This is the value specified in the *local config file*.
- **`home`:** Running user's home directory (on Unix and Windows, see Python's `os.path.expanduser()`).
- **`test_data_dir`:** Directory in Pimlico distribution where test data is stored (`test/data` in Pimlico root). Used in test pipelines, which take all their input data from this directory.

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

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

Directives

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

- **`variant:`**

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

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

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

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

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

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

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

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

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

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

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

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

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

Multiple parameter values

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

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

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

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

For example:

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

Tying alternatives

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

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

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

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

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

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

Naming alternatives

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

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

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

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

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

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

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

Expanding alternatives down the pipeline

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

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

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

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

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

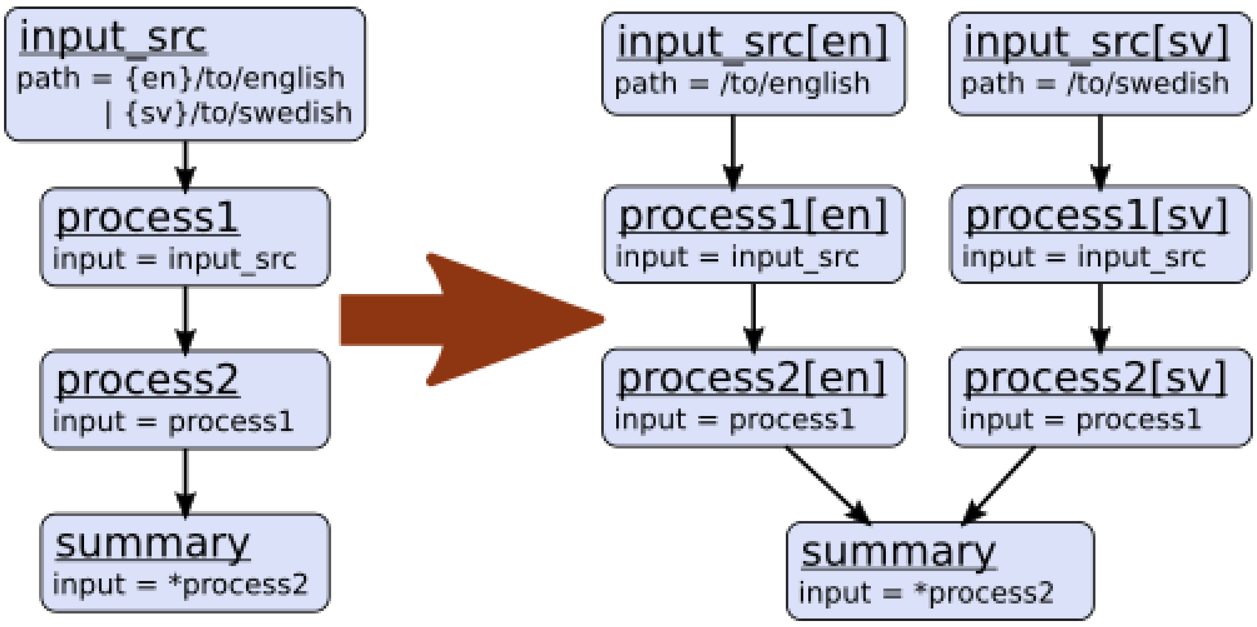
Module variables are set by including parameters in a module's config of the form `modvar_<name> = <value>`. This will assign `value` to the variable `name` for this module. The simplest form of assignment is just a string literal, enclosed in double quotes:

```
[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

Todo: Document variants

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 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.

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

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

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

Defining module dependencies

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

Some examples

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

1.2.6 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 *Long-term and short-term 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.

Long-term and short-term 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.

The **short-term store** should be on a disk that's as fast as possible to write to. For example, avoid using an NFS disk. It needs to be large enough to store output between pipeline stages, though you can easily move output from earlier stages into the long-term store as you go along.

The **long-term store** is where things are typically put at the end of a pipeline. It therefore doesn't need to be super-fast to access, but you may want it to be in a location that gets backed up, so you don't lose your valuable output.

For a simple setup, these could be just two subdirectories of the same directory, or actually the same directory. However, it can be useful to distinguish them.

Specific the locations in the local config like this:

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

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

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:

1.2.7 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 CAEVO event extractor

Path	pimlico.modules.caevo
Executable	yes

CAEVO is Nate Chambers' CAscading EVent Ordering system, a tool for extracting events of many types from text and ordering them.

CAEVO is open source, implemented in Java, so is easily integrated into Pimlico using Py4J.

Inputs

Name	Type(s)
documents	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
events	*CaevoCorpus*

Options

Name	Description	Type
sieves	Filename of sieve list file, or path to the file. If just a filename, assumed to be in Caevo model dir (models/caevo). Default: default.sieves (supplied with Caevo)	string

Submodules

CAEVO output convertor

Path	pimlico.modules.caevo.output
Executable	yes

Tool to split up the output from Caevo and convert it to other datatypes.

Caevo output includes the output from a load of NLP tools that it runs as prerequisites to event extraction, etc. The individual parts of the output can easily be retrieved from the output corpus via the output datatype. In order to be able to use them as input to other modules, they need to be converted to compatible standard datatypes.

For example, tokenization output is stored in Caevo's XML output using a special format. Instead of writing other modules in such a way as to be able to pull this information out of the :class:~pimlico.datatypes.CaevoCorpus, you can filter the output using this module to provide a :class:~pimlico.datatypes.TokenizedCorpus, which is a standard format for input to other module types.

As with other document map modules, you can use this as a filter (*filter=T*), so you can actually need to commit the converted data to disk.

Todo: Add more output convertors: currently only provides tokenization

Inputs

Name	Type(s)
documents	*CaevoCorpus*

Outputs

No non-optional outputs

Optional

Name	Type(s)
tokenized	*TokenizedCorpus*
parse	*ConstituencyParseTreeCorpus*
pos	WordAnnotationCorpusWithWordAndPos

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool

1.3.2 C&C parser

Path	pimlico.modules.candc
Executable	yes

Wrapper around the original C&C parser.

Takes tokenized input and parses it with C&C. The output is written exactly as it comes out from C&C. It contains both GRs and supertags, plus POS-tags, etc.

The wrapper uses C&C's SOAP server. It sets the SOAP server running in the background and then calls C&C's SOAP client for each document. If parallelizing, multiple SOAP servers are set going and each one is kept constantly fed with documents.

Inputs

Name	Type(s)
documents	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
parsed	`CandcOutputCorpus`

Options

Name	Description	Type
model	Absolute path to models directory or name of model set. If not an absolute path, assumed to be a subdirectory of the candcs models dir (see instructions in models/candc/README on how to fetch pre-trained models)	string

1.3.3 Stanford CoreNLP

Path	pimlico.modules.corenlp
Executable	yes

Process documents one at a time with the Stanford CoreNLP toolkit. CoreNLP provides a large number of NLP tools, including a POS-tagger, various parsers, named-entity recognition and coreference resolution. Most of these tools can be run using this module.

The module uses the CoreNLP server to accept many inputs without the overhead of loading models. If parallelizing, only a single CoreNLP server is run, since this is designed to set multiple Java threads running if it receives multiple queries at the same time. Multiple Python processes send queries to the server and process the output.

The module has no non-optional outputs, since what sort of output is available depends on the options you pass in: that is, on which tools are run. Use the annotations option to choose which word annotations are added. Otherwise, simply select the outputs that you want and the necessary tools will be run in the CoreNLP pipeline to produce those outputs.

Currently, the module only accepts tokenized input. If pre-POS-tagged input is given, for example, the POS tags won't be handed into CoreNLP. In the future, this will be implemented.

We also don't currently provide a way of choosing models other than the standard, pre-trained English models. This is a small addition that will be implemented in the future.

Inputs

Name	Type(s)
documents	TarredCorpus<WordAnnotationsDocumentType|TokenizedDocumentType|RawTextDocumentType>

Outputs

No non-optional outputs

Optional

Name	Type(s)
annotations	*AnnotationFieldsFromOptions*
tokenized	*TokenizedCorpus*
parse	*ConstituencyParseTreeCorpus*
parse-deps	*StanfordDependencyParseCorpus*
dep-parse	*StanfordDependencyParseCorpus*
raw	*JsonDocumentCorpus*
coref	*CorefCorpus*

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
timeout	Timeout for the CoreNLP server, which is applied to every job (document). Number of seconds. By default, we use the server's default timeout (15 secs), but you may want to increase this for more intensive tasks, like coref	float
readable	If True, JSON outputs are formatted in a readable fashion, pretty printed. Otherwise, they're as compact as possible. Default: False	bool
annotators	Comma-separated list of word annotations to add, from CoreNLP's annotators. Choose from: word, pos, lemma, ner	string
dep_type	Type of dependency parse to output, when outputting dependency parses, either from a constituency parse or direct dependency parse. Choose from the three types allowed by CoreNLP: 'basic', 'collapsed' or 'collapsed-ccprocessed'	'basic', 'collapsed' or 'collapsed-ccprocessed'

1.3.4 Corpus manipulation

Core modules for generic manipulation of mainly iterable corpora.

Corpus concatenation

Path	pimlico.modules.corpora.concat
Executable	no

Concatenate two corpora to produce a bigger corpus.

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 *`IterableCorpus`*

Outputs

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

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	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
stats	*NamedFile()*

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	*TarredCorpus*

Outputs

Name	Type(s)
formatted	*TarredCorpus*

Options

Name	Description	Type
for-mat-ter	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

Corpus document list filter

Path	pimlico.modules.corpora.list_filter
Executable	yes

Similar to :mod:pimlico.modules.corpora.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	`TarredCorpus`
list	`StringList`

Outputs

Name	Type(s)
set1	`same as input corpus`
set2	`same as input corpus`

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	`TarredCorpus`

Outputs

Name	Type(s)
set1	`same as input corpus`
set2	`same as input corpus`
doc_list1	`StringList`

Optional

Name	Type(s)
doc_list2	*StringList*

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

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)
documents	*IterableCorpus*

Outputs

Name	Type(s)
documents	`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
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
size	(required) Number of documents to include	int

Tar archive grouper

Path	pimlico.modules.corpora.tar
Executable	yes

Group the files of a multi-file iterable corpus into tar 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 files are simply grouped linearly into a series of tar archives such that each (apart from the last) contains the given number.

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: There is a fundamental problem with this module. It stores the raw data that it gets as input, and reports the output type as the same as the input type. However, it doesn't correctly write that type. A lot of the time, this isn't a problem, but it means that it doesn't write corpus metadata that may be needed by the datatype to read the documents correctly.

The new datatypes system will provide a solution to this problem, but until then the safest approach is not to use this module, but always use `tar_filter` instead, which doesn't have this problem.

Inputs

Name	Type(s)
documents	*IterableCorpus*

Outputs

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

Options

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

Tar archive grouper (filter)

Path	pimlico.modules.corpora.tar_filter
Executable	no

Like `tar`, but doesn't write the archives to disk. Instead simulates the behaviour of tar but as a filter, grouping files on the fly and passing them through with an archive name

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	`IterableCorpus`

Outputs

Name	Type(s)
documents	`tarred corpus with input doc type`

Options

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

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	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
vocab	*Dictionary*

Options

Name	Description	Type
threshold	Minimum number of occurrences required of a term to be included	int
max_prop	Include terms that occur in max this proportion of documents	float
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
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

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	TarredCorpus<IntegerListsDocumentType>
vocab	*Dictionary*

Outputs

Name	Type(s)
distribution	*NumpyArray*

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

Tokenized corpus to ID mapper

Path	pimlico.modules.corpora.vocab_mapper
Executable	yes

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType>
vocab	*Dictionary*

Outputs

Name	Type(s)
ids	*IntegerListsDocumentCorpus*

Options

Name	Description	Type
oov	If given, special token to map all OOV characters to. Otherwise, use vocab_size+1 as index	string

1.3.5 Embedding feature extractors and trainers

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.

Dependency feature extractor for embeddings

Path	pimlico.modules.embeddings.dependencies
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
dependencies	TarredCorpus<CoNLLDependencyParseDocumentType>

Outputs

Name	Type(s)
term_features	*TermFeatureListCorpus*

Options

Name	Description	Type
lemma	Use lemmas as terms instead of the word form. Note that if you didn't run a lemmatizer before dependency parsing the lemmas are probably actually just copies of the word forms	bool
con-dense_prep	Where a word is modified . . . TODO	string
term_pos	Only extract features for terms whose POSs are in this comma-separated list. Put a * at the end to denote POS prefixes	comma-separated list of strings
skip_types	Dependency relations to skip, separated by commas	comma-separated list of strings

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's Python (+C) implementation of word2vec on a Pimlico corpus.

Inputs

Name	Type(s)
text	TarredCorpus<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
nega-tive_samples	number of negative samples to include per positive. Default: 5	int
size	number of dimensions in learned vectors. Default: 200	int

1.3.6 Feature set processing

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

Key-value to term-feature converter

Path	pimlico.modules.features.term_feature_compiler
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
key_values	TarredCorpus<KeyValueListDocumentType>

Outputs

Name	Type(s)
term_features	*TermFeatureListCorpus*

Options

Name	Description	Type
term_keys	Name of keys (feature names in the input) which denote terms. The first one found in the keys of a particular data point will be used as the term for that data point. Any other matches will be removed before using the remaining keys as the data point's features. Default: just 'term'	comma-separated list of strings
include_feature_keys	If True, include the key together with the value from the input key-value pairs as feature names in the output. Otherwise, just use the value. E.g. for input [prop=wordy, poss=my], if True we get features [prop_wordy, poss_my] (both with count 1); if False we get just [wordy, my]. Default: False	bool

Term-feature matrix builder

Path	pimlico.modules.features.term_feature_matrix_builder
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
data	*IndexedTermFeatureListCorpus*

Outputs

Name	Type(s)
matrix	*ScipySparseMatrix*

Term-feature corpus vocab builder

Path	pimlico.modules.features.vocab_builder
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
term_features	TarredCorpus<TermFeatureListDocumentType>

Outputs

Name	Type(s)
term_vocab	*Dictionary*
feature_vocab	*Dictionary*

Options

Name	Description	Type
feature_limit	Limit vocab size to this number of most common entries (after other filters)	int
feature_max_prop	Include features that occur in max this proportion of documents	float
term_max_prop	Include terms that occur in max this proportion of documents	float
term_threshold	Minimum number of occurrences required of a term to be included	int
feature_threshold	Minimum number of occurrences required of a feature to be included	int
term_limit	Limit vocab size to this number of most common entries (after other filters)	int

Term-feature corpus vocab mapper

Path	pimlico.modules.features.vocab_mapper
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
data	TarredCorpus<TermFeatureListDocumentType>
term_vocab	`Dictionary`
feature_vocab	`Dictionary`

Outputs

Name	Type(s)
data	*`IndexedTermFeatureListCorpus`*

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.

Note: These input readers are ultimately intended to replace reading input data using a datatype's `input_module_options`. That functionality will be removed altogether as part of the development of the new datatype system, so it should be phased out now and replaced by input reader modules for each datatype.

Package pimlico.modules.input.embeddings

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
path	(required) Path to the FastText embedding file	string
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

FastText embedding reader using 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.

Inputs

No inputs

Outputs

Name	Type(s)
embeddings	*Embeddings*

Options

Name	Description	Type
path	(required) Path to the FastText embedding file (.bin)	string

Text corpora

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.

See also:

Datatype ***`pimlico.datatypes.files.UnnamedFileCollection`*** The datatype previously used for reading in file collections, now being phased out to be replaced by this input reader.

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	`OutputType`

Options

Name	Description	Type
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
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	comma-separated list of strings
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
encoding	Encoding to assume for input files. Default: utf8	string

Package pimlico.modules.input.text_annotations

VRT annotated text files

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

Input reader for VRT text collections (VeRticalized Text, as used by Korp:). Reads in files from arbitrary locations in the same way as `pimlico.modules.input.text.raw_text_files`.

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	`VRTOutputType`

Options

Name	Description	Type
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
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	comma-separated list of strings
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
encoding	Encoding to assume for input files. Default: utf8	string

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

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	`VRTTextOutputType`

Options

Name	Description	Type
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
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	comma-separated list of strings
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
encoding	Encoding to assume for input files. Default: utf8	string

XML documents

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.

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	`XMLOutputType`

Options

Name	Description	Type
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	comma-separated list of strings
encoding	Encoding to assume for input files. Default: utf8	string
document_node_type	XML node type to extract documents from (default: 'doc')	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
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 strings
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
exclude	A list of files to exclude. Specified in the same way as *files* (except without line ranges). This allows you to specify a glob in *files* and then exclude individual files from it (you can use globs here too)	comma-separated list of strings

1.3.8 Malt dependency parser

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

Annotated text to CoNLL dep parse input converter

Path	pimlico.modules.malt.conll_parser_input
Executable	yes

Converts word-annotations to CoNLL format, ready for input into the Malt parser. Annotations must contain words and POS tags. If they contain lemmas, all the better; otherwise the word will be repeated as the lemma.

Inputs

Name	Type(s)
annotations	`WordAnnotationCorpus` with 'word' and 'pos' fields

Outputs

Name	Type(s)
conll_data	*CoNLLDependencyParseInputCorpus*

Malt dependency parser

Path	pimlico.modules.malt.parse
Executable	yes

Todo: Document this module

Todo: Replace check_runtime_dependencies() with get_software_dependencies()

Inputs

Name	Type(s)
documents	TarredCorpus<CoNLLDependencyParseDocumentType>

Outputs

Name	Type(s)
parsed	*CoNLLDependencyParseCorpus*

Options

Name	Description	Type
model	Filename of parsing model, or path to the file. If just a filename, assumed to be Malt models dir (models/malt). Default: engmalt.linear-1.7.mco, which can be acquired by 'make malt' in the models dir	string
no_gzip	By default, we gzip each document in the output data. If you don't do this, the output can get very large, since it's quite a verbose output format	bool

1.3.9 OpenNLP modules

A collection of module types to wrap individual OpenNLP tools.

OpenNLP coreference resolution

Path	pimlico.modules.opennlp.coreference
Executable	yes

Todo: Document this module

Todo: Replace check_runtime_dependencies() with get_software_dependencies()

Use local config setting opennlp_memory to set the limit on Java heap memory for the OpenNLP processes. If parallelizing, this limit is shared between the processes. That is, each OpenNLP worker will have a memory limit of *opennlp_memory / processes*. That setting can use *g*, *G*, *m*, *M*, *k* and *K*, as in the Java setting.

Inputs

Name	Type(s)
parses	TarredCorpus<TreeStringsDocumentType>

Outputs

Name	Type(s)
coref	`CorefCorpus`

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
model	Coreference resolution model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/). Default: ‘’ (standard English opennlp model in models/opennlp/)	string
read-able	If True, pretty-print the JSON output, so it’s human-readable. Default: False	bool
time-out	Timeout in seconds for each individual coref resolution task. If this is exceeded, an InvalidDocument is returned for that document	int

OpenNLP coreference resolution

Path	pimlico.modules.opennlp.coreference_pipeline
Executable	yes

Runs the full coreference resolution pipeline using OpenNLP. This includes sentence splitting, tokenization, pos tagging, parsing and coreference resolution. The results of all the stages are available in the output.

Use local config setting opennlp_memory to set the limit on Java heap memory for the OpenNLP processes. If parallelizing, this limit is shared between the processes. That is, each OpenNLP worker will have a memory limit of *opennlp_memory / processes*. That setting can use *g*, *G*, *m*, *M*, *k* and *K*, as in the Java setting.

Inputs

Name	Type(s)
text	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
coref	`CorefCorpus`

Optional

Name	Type(s)
tokenized	`TokenizedCorpus`
pos	`WordAnnotationCorpusWithPos`
parse	`ConstituencyParseTreeCorpus`

Options

Name	Description	Type
gzip	If True, each output, except annotations, for each document is gzipped. This can help reduce the storage occupied by e.g. parser or coref output. Default: False	bool
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
parse_model	Parser model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/)	string
timeout	Timeout in seconds for each individual coref resolution task. If this is exceeded, an InvalidDocument is returned for that document	int
coref_model	Coreference resolution model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/). Default: ‘’ (standard English opennlp model in models/opennlp/)	string
readable	If True, pretty-print the JSON output, so it’s human-readable. Default: False	bool
pos_model	POS tagger model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string
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

OpenNLP NER

Path	pimlico.modules.opennlp.ner
Executable	yes

Named-entity recognition using OpenNLP's tools.

By default, uses the pre-trained English model distributed with OpenNLP. If you want to use other models (e.g. for other languages), download them from the OpenNLP website to the models dir (*models/opennlp*) and specify the model name as an option.

Note that the default model is for identifying person names only. You can identify other name types by loading other pre-trained OpenNLP NER models. Identification of multiple name types at the same time is not (yet) implemented.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType|WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	`SentenceSpansCorpus`

Options

Name	Description	Type
model	NER model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string

OpenNLP constituency parser

Path	pimlico.modules.opennlp.parse
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
documents	TarredCorpus<TokenizedDocumentType> or `WordAnnotationCorpus` with 'word' field

Outputs

Name	Type(s)
parser	*ConstituencyParseTreeCorpus*

Options

Name	Description	Type
model	Parser model, full path or directory name. If a filename is given, it is expected to be in the OpenNLP model directory (models/opennlp/)	string

OpenNLP POS-tagger

Path	pimlico.modules.opennlp.pos
Executable	yes

Part-of-speech tagging using OpenNLP's tools.

By default, uses the pre-trained English model distributed with OpenNLP. If you want to use other models (e.g. for other languages), download them from the OpenNLP website to the models dir (*models/opennlp*) and specify the model name as an option.

Inputs

Name	Type(s)
text	TarredCorpus<TokenizedDocumentType|WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	*AddAnnotationField*

Options

Name	Description	Type
model	POS tagger model, full path or filename. If a filename is given, it is expected to be in the opennlp model directory (models/opennlp/)	string

OpenNLP tokenizer

Path	pimlico.modules.opennlp.tokenize
Executable	yes

Sentence splitting and tokenization using OpenNLP's tools.

Inputs

Name	Type(s)
text	TarredCorpus<RawTextDocumentType>

Outputs

Name	Type(s)
documents	*TokenizedCorpus*

Options

Name	Description	Type
to-ken_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
tok-enize_only	By default, sentence splitting is performed prior to tokenization. If tokenize_only is set, only the tokenization step is executed	bool
sen-tence_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

1.3.10 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.

R script executor

Path	pimlico.modules.r.script
Executable	yes

Simple interface to R that just involves running a given R script, first substituting in some paths from the pipeline, making it easy to pass in data from the output of other modules.

Inputs

Name	Type(s)
sources	*list* of *PimlicoDatatype*

Outputs

Name	Type(s)
output	*NamedFile()*

Options

Name	Description	Type
script	(required) Path to the script to be run. The script itself may include substitutions of the form '{{inputX}}', which will be replaced with the absolute path to the data dir of the Xth input, and '{{output}}', which will be replaced with the absolute path to the output dir. The latter allows the script to output things other than the output file, which always exists and contains the full script's output	string

1.3.11 Regular expressions

Regex annotated text matcher

Path	pimlico.modules.regex.annotated_text
Executable	yes

Todo: Document this module

Inputs

Name	Type(s)
documents	TarredCorpus<WordAnnotationsDocumentType>

Outputs

Name	Type(s)
documents	*KeyValueListCorpus*

Options

Name	Description	Type
expr	(required) An expression to determine what to search for in sentences. Consists of a sequence of tokens, each matching one field in the corresponding token's annotations in the data. These are specified in the form field[x], where field is the name of a field supplied by the input data and x is the value required of that field. If x ends in a *, it will match prefixes: e.g. pos[NN*]*. If no field name is given, the default 'word' is used. A token of the form 'x=y' matches the expression y as above and assigns the matching word to the extracted variable x (to be output). You may also extract a different annotation field by specifying x=f:y, where f is the field name to be extracted. E.g. 'what a=lemma:pos[NN*] lemma[come] with b=pos[NN*]' matches phrases like 'what meals come with fries', producing 'a=meal' and 'b=fries'. Both pos and lemma need to be fields in the dataset'. If you give multiple whole expressions separated by |s, matches will be collected from all of them	string

1.3.12 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 matrix factorization

Path	pimlico.modules.sklearn.matrix_factorization
Executable	yes

Provides a simple interface to Scikit-Learn's various matrix factorization models.

Since they provide a consistent training interface, you can simply choose the class name of the method you want to use and specify options relevant to that method in the `options` option. For available options, take a look at the table of parameters in the Scikit-Learn documentation for each class.

Inputs

Name	Type(s)
matrix	`ScipySparseMatrix`

Outputs

Name	Type(s)
w	`NumpyArray`
h	`NumpyArray`

Options

Name	Description	Type
class	(required) Scikit-learn class to use to fit the matrix factorization. Should be the name of a class in the package sklearn.decomposition that has a fit_transform() method and a **components_** attribute. Supported classes: NMF, SparsePCA, ProjectedGradientNMF, FastICA, FactorAnalysis, PCA, RandomizedPCA, LatentDirichletAllocation, TruncatedSVD	'NMF', 'SparsePCA', 'ProjectedGradientNMF', 'FastICA', 'FactorAnalysis', 'PCA', 'RandomizedPCA', 'LatentDirichletAllocation' or 'TruncatedSVD'
options	Options to pass into the constructor of the sklearn class, formatted as a JSON dictionary (potentially without the {}s). E.g.: 'n_components=200, solver="cd", tol=0.0001, max_iter=200'	string

1.3.13 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	TarredCorpus<TextDocumentType>

Outputs

Name	Type(s)
corpus	`CharacterTokenizedDocumentTypeTarredCorpus`

Normalize tokenized text

Path	pimlico.modules.text.normalize
Executable	yes

Perform text normalization on tokenized documents.

Currently, this includes only case normalization (to upper or lower case). In the future, more normalization operations may be added.

Inputs

Name	Type(s)
corpus	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
corpus	*TokenizedCorpus*

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 ''

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	TarredCorpus<TextDocumentType>

Outputs

Name	Type(s)
corpus	`TokenizedDocumentTypeTarredCorpus`

Options

Name	Description	Type
splitter	Character or string to split on. Default: space	<type 'unicode'>

Tokenized text to text

Path	pimlico.modules.text.untokenize
Executable	yes

Filter to take tokenized text and join it together to make raw text.

This module shouldn't be necessary and will be removed later. For the time being, it's here as a workaround for [this problem](https://github.com/markgw/pimlico/issues/1#issuecomment-383620759), until it's solved in the datatype redesign.

Tokenized text is a subtype of text, so theoretically it should be acceptable to modules that expect plain text (and is considered so by typechecking). But it provides an incompatible data structure, so things go bad if you use it like that.

Inputs

Name	Type(s)
corpus	TarredCorpus<TokenizedDocumentType>

Outputs

Name	Type(s)
corpus	`TextDocumentTypeTarredCorpus`

Options

Name	Description	Type
sentence_joiner	String to join lines/sentences on. (Default: linebreak)	<type 'unicode'>
joiner	String to join words on. (Default: space)	<type 'unicode'>

1.3.14 General utilities

General utilities for things like filesystem manipulation.

Module output alias

Path	pimlico.modules.utility.alias
Executable	no

Alias a datatype coming from the output of another module.

Used to assign a handy identifier to the output of a module, so that we can just refer to this alias module later in the pipeline and use its default output. This can help make for a more readable pipeline config.

For example, say we use `split` to split a dataset into two random subsets. The two splits can be accessed by referring to the two outputs of that module: *split_module.set1* and *split_module.set2*. However, it's easy to lose track of what these splits are supposed to be used for, so we might want to give them names:

```
[split_module]
type=pimlico.modules.corpora.split
set1_size=0.2

[test_set]
type=pimlico.modules.utility.alias
input=split_module.set1

[training_set]
type=pimlico.modules.utility.alias
input=split_module.set2

[training_routine]
type=...
input_corpus=training_set
```

Note the difference between using this module and using the special *alias* module type. The *alias* type creates an alias for a whole module, allowing you to refer to all of its outputs, inherit its settings, and anything else you could do with the original module name. This module, however, provides an alias for exactly one output of a module and generates a module instance of its own in the pipeline (albeit a filter module).

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

Inputs

Name	Type(s)
input	`PimlicoDatatype`

Outputs

Name	Type(s)
output	`same as input corpus`

Copy file

Path	pimlico.modules.utility.copy_file
Executable	yes

Copy a file

Simple utility for copying a file (which presumably comes from the output of another module) into a particular location. Useful for collecting together final output at the end of a pipeline.

Inputs

Name	Type(s)
source	`list` of `File`

Outputs

Name	Type(s)
documents	`TarredCorpus`

Options

Name	Description	Type
tar-get_name	Name to rename the target file to. If not given, it will have the same name as the source file. Ignored if there's more than one input file	string
tar-get_dir	(required) Path to directory into which the file should be copied. Will be created if it doesn't exist	string

1.3.15 Visualization tools

Modules for plotting and suchlike

Bar chart plotter

Path	pimlico.modules.visualization.bar_chart
Executable	yes

Inputs

Name	Type(s)
values	`list` of `NumericResult`

Outputs

Name	Type(s)
plot	`PlotOutput`

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.

Inputs

Name	Type(s)
vectors	`list` of `Embeddings`

Outputs

Name	Type(s)
plot	`PlotOutput`

Options

Name	Description	Type
skip	Number of most frequent words to skip, taking the next most frequent after these. Default: 0	int
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'
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	comma-separated list of strings
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
words	Number of most frequent words to plot. Default: 50	int

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
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
longstore	Move a particular module's output from the short-term store to the long-term store
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
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
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.

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 browse

Command-line tool subcommand

View the data output by a module.

Usage:

```
pimlico.sh [...] browse module_name [output_name] [-h] [--raw] [--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
`--raw, -r`	Don't parse the data using the output datatype (i.e. just read the raw text). If not set, we output the result of applying unicode() to the parsed data structure, or a custom formatting if the datatype loaded defines one
`--skip-invalid`	Skip over invalid documents, instead of showing the error that caused them to be invalid
`--formatter, -f`	Fully qualified class name of a subclass of DocumentBrowserFormatter to use to determine what to output for each document. If specified, –raw is ignored. You may also choose from the named standard formatters for the datatype in question. Use '-f help' to see a list of available formatters

1.4.5 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

1.4.6 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

1.4.7 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

1.4.8 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]
```

1.4.9 longstore

Command-line tool subcommand

Move a particular module's output from the short-term store to the long-term store. It will still be found here by input readers. You might want to do this if your long-term store is bigger, to keep down the short-term store size.

Usage:

```
pimlico.sh [...] longstore [modules [modules ...]] [-h]
```

Positional arguments

Arg	Description
`[modules [modules ...]]`	The names (or numbers) of the module whose output to move

1.4.10 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

1.4.11 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]
```

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

1.4.12 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

1.4.13 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

1.4.14 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

1.4.15 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

1.4.16 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

1.4.17 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]
```

1.4.18 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

1.4.19 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

Submodules

pimlico.cli.browser.formatter module

The command-line iterable corpus browser displays one document at a time. It can display the raw data from the corpus files, which sometimes is sufficiently human-readable to not need any special formatting. It can also parse the data using its datatype and output text either from the datatype's standard unicode representation or, if the document datatype provides it, a special browser formatting of the data.

When viewing output data, particularly during debugging of modules, it can be useful to provide special formatting routines to the browser, rather than using or overriding the datatype's standard formatting methods. For example, you might want to pull out specific attributes for each document to get an overview of what's coming out.

The browser command accepts a command-line option that specifies a Python class to format the data. This class should be a subclass of :class:~pimlico.cli.browser.formatter.DocumentBrowserFormatter that accepts a datatype compatible with the datatype being browsed and provides a method to format each document. You can write these in your custom code and refer to them by their fully qualified class name.

class DocumentBrowserFormatter(*corpus*)

Bases: `object`

Base class for formatters used to post-process documents for display in the iterable corpus browser.

DATATYPE

alias of *pimlico.datatypes.documents.DataPointType*

RAW_INPUT = False

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

filter_document(*doc*)
Each doc is passed through this function directly after being read from the corpus. If None is returned, the doc is skipped. Otherwise, the result is used instead of the doc data. The default implementation does nothing.

class DefaultFormatter(*corpus*, *raw_data=False*)
Bases: *pimlico.cli.browser.formatter.DocumentBrowserFormatter*

Generic implementation of a browser formatter that's used if no other formatter is given.

DATATYPE
alias of *pimlico.datatypes.base.IterableCorpus*

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class InvalidDocumentFormatter(*corpus*)
Bases: *pimlico.cli.browser.formatter.DocumentBrowserFormatter*

Formatter that skips over all docs other than invalid results. Uses standard formatting for InvalidDocument information.

DATATYPE
alias of *pimlico.datatypes.base.IterableCorpus*

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

filter_document(*doc*)
Each doc is passed through this function directly after being read from the corpus. If None is returned, the doc is skipped. Otherwise, the result is used instead of the doc data. The default implementation does nothing.

typecheck_formatter(*formatted_doc_type*, *formatter_cls*)
Check that a document type is compatible with a particular formatter.

load_formatter(*dataset*, *formatter_name=None*, *parse=True*)
Load a formatter specified by its fully qualified Python class name. If None, loads the default formatter. You may also specify a formatter by name, choosing from one of the standard ones that the formatted datatype gives.

Parameters

- **formatter_name** – class name, or class
- **dataset** – dataset that will be formatted
- **parse** – only used if the default formatter is loaded, determines *raw_data* (= *not parse*)

Returns instantiated formatter

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

browse_data(*data*, *formatter*, *parse=False*, *skip_invalid=False*)

class CorpusState(*corpus*)
    Bases: `object`

    Keep track of which document we're on.

    next_document()

    skip(*n*)

class InputDialog(*text*, *input_edit*)
    Bases: `urwid.widget.WidgetWrap`

    A dialog that appears with an input

    signals = ['close', 'cancel']

    keypress(*size*, *k*)

class MessageDialog(*text*, *default=None*)
    Bases: `urwid.widget.WidgetWrap`

    A dialog that appears with a message

class InputPopupLauncher(*original_widget*, *text*, *input_edit*, *callback=None*)
    Bases: `urwid.wimp.PopUpLauncher`

    create_pop_up()
        Subclass must override this method and return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

    get_pop_up_parameters()
        Subclass must override this method and have it return a dict, eg:

        {'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

        This method is called each time this widget is rendered.

skip_popup_launcher(*original_widget*, *text*, *default=None*, *callback=None*)

save_popup_launcher(*original_widget*, *text*, *default=None*, *callback=None*)

class MessagePopupLauncher(*original_widget*, *text*)
    Bases: `urwid.wimp.PopUpLauncher`

    create_pop_up()
        Subclass must override this method and return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

    get_pop_up_parameters()
        Subclass must override this method and have it return a dict, eg:

        {'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

        This method is called each time this widget is rendered.

Module contents

pimlico.cli.debug package

Submodules

pimlico.cli.debug.stepper module

class Stepper

Bases: `object`

Type that stores the state of the stepping process. This allows information and parameters to be passed around through the process and updated as we go. For example, if particular type of output is disabled by the user, a parameter can be updated here so we know not to output it later.

enable_step_for_pipeline(*pipeline*)

Prepares a pipeline to run in step mode, modifying modules and wrapping methods to supply the extra functionality.

This approach means that we don't have to consume extra computation time checking whether step mode is enabled during normal runs.

Parameters **pipeline** – instance of PipelineConfig

instantiate_output_datatype_decorator(*instantiate_output_datatype*, *module_name*, *output_names*, *stepper*)

wrap_tarred_corpus(*dtype*, *module_name*, *output_name*, *stepper*)

archive_iter_decorator(*archive_iter*, *module_name*, *output_name*, *stepper*)

get_input_decorator(*get_input*, *module_name*, *stepper*)

Decorator to wrap a module info's get_input() method so when know where inputs are being used.

option_message(*message_lines*, *stepper*, *options=None*, *stack_trace_option=True*, *category=None*)

Module contents

Extra-verbose debugging facility

Tools for very slowly and verbosely stepping through the processing that a given module does to debug it.

Enabled using the *–step* switch to the run command.

fmt_frame_info(*info*)

output_stack_trace(*frame=None*)

pimlico.cli.shell package

Submodules

pimlico.cli.shell.base module

class ShellCommand

Bases: `object`

Base class used to provide commands for exploring a particular datatype. A basic set of commands is provided for all datatypes, but specific datatype classes may provide their own, by overriding the *shell_commands* attribute.

commands = []

help_text = None

execute (*shell*, **args*, ***kwargs*)

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*)

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*)

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

class InstallCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Install missing dependencies.

command_name = 'install'

command_help = 'Install missing module library dependencies'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class DepsCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Output information about module dependencies.

command_name = 'deps'

command_help = "List information about software dependencies: whether they're availab

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

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 the

command_desc = 'Remove all module output directories that do not correspond to a module

run_command(*pipeline*, *opts*)

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

command_desc = 'Dump the entire available output data from a given pipeline module to

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 d

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

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 datas

command_desc = 'Show the (expected) locations of the inputs of a given module'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class OutputCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'output'

command_help = "Show the location where the given module's output data will be (or has

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

pimlico.cli.main module

Main command-line script for running Pimlico, typically called from *pimlico.sh*.

Provides access to many subcommands, acting as the primary interface to Pimlico's functionality.

class VariantsCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'variants'

command_help = 'List the available variants of a pipeline config'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class LongStoreCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'longstore'

command_help = "Move a particular module's output from the short-term store to the lon

command_desc = "Move a particular module's output from the short-term store to the lon

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

class UnlockCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Forcibly remove an execution lock from a module. If a lock has ended up getting left on when execution exited prematurely, use this to remove it.

When a module starts running, it is locked to avoid making a mess of your output data by running the same module from another terminal, or some other silly mistake (I know, for some of us this sort of behaviour is frustratingly common).

Usually shouldn't be necessary, even if there's an error during execution, since the module should be unlocked when Pimlico exits, but occasionally (e.g. if you have to forcibly kill Pimlico during execution) the lock gets left on.

command_name = 'unlock'

command_help = "Forcibly remove an execution lock from a module. If a lock has ended u

command_desc = 'Forcibly remove an execution lock from a module'

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

class BrowseCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'browse'

command_help = 'View the data output by a module'

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

class VisualizeCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'visualize'

command_help = '(Not yet fully implemented!) Visualize the pipeline, with status infor

command_desc = 'Comming soon...visualize the pipeline in a pretty way'

add_arguments (*parser*)

run_command (*pipeline*, *opts*)

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: `object`

A class used as a static global data structure to provide access to the loaded pipeline when running the Pimlico Python shell command.

This should never be used in any other context to pass around loaded pipelines or other global data. We don't do that sort of thing.

class PythonShellCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

command_name = 'python'

command_help = 'Load the pipeline config and enter a Python interpreter with access to

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

get_pipeline()

This function may be used in scripts that are expected to be run exclusively from the Pimlico Python shell command (`python`) to get hold of the pipeline that was specified on the command line and loaded when the shell was started.

exception ShellContextError

Bases: `exceptions.Exception`

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

class RunCmd

Bases: *pimlico.cli.subcommands.PimlicoCLISubcommand*

Main command for executing Pimlico modules from the command line *run* command.

command_name = 'run'

command_help = 'Execute an individual pipeline module, or a sequence'

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

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: `object`

Base class for defining subcommands to the main command line tool.

This allows us to split up subcommands, together with all their arguments/options and their functionality, since there are quite a lot of them.

Documentation of subcommands should be supplied in the following ways:

- Include help texts for positional args and options in the add_arguments() method. They will all be included in the doc page for the command.
- Write a very short description of what the command is for (a few words) in command_desc. This will be used in the summary table / TOC in the docs.
- Write a short description of what the command does in `command_help`. This will be available in command-line help and used as a fallback if you don't do the next point.
- Write a good guide to using the command (or at least say what it does) in the class' docstring (i.e. overriding this). This will form the bulk of the command's doc page.

command_name = None

command_help = None

command_desc = None

add_arguments(*parser*)

run_command(*pipeline*, *opts*)

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: `object`

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.

dependencies()

Returns a list of instances of :class:SoftwareDependency subcalsses 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 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<virtua

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

class JavaDependency(*name*, *classes=[]*, *jars=[]*, *class_dirs=[]*, ***kwargs*)

Bases: `pimlico.core.dependencies.base.SoftwareDependency`

Base class for Java library dependencies.

In addition to the usual functionality provided by dependencies, subclasses of this provide contributions to the Java classpath in the form of directories of jar files.

The instance has a set of representative Java classes that the checker will try to load to check whether the library is available and functional. It will also check that all jar files exist.

Jar paths and class directory paths are assumed to be relative to the Java lib dir (lib/java), unless they are absolute paths.

Subclasses should provide install() and override installable() if it's possible to install them automatically.

problems(*local_config*)

Returns a list of problems standing in the way of the dependency being available. If the list is empty, the dependency is taken to be installed and ready to use.

Overriding methods should call super method.

installable()

Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

get_classpath_components()

class JavaJarsDependency(*name*, *jar_urls*, ***kwargs*)

Bases: `pimlico.core.dependencies.java.JavaDependency`

Simple way to define a Java dependency where the library is packaged up in a jar, or a series of jars. The jars should be given as a list of (name, url) pairs, where name is the filename the jar should have and url is a url from which it can be downloaded.

URLs may also be given in the form "url->member", where url is a URL to a tar.gz or zip archive and member is a member to extract from the archive. If the type of the file isn't clear from the URL (i.e. if it doesn't have ".zip" or ".tar.gz" in it), specify the intended extension in the form "[ext]url->member", where ext is "tar.gz" or "zip".

installable()

Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install(*local_config*, *trust_downloaded_archives=False*)

Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

class PimlicoJavaLibrary(*name*, *classes=[]*, *additional_jars=[]*)

Bases: `pimlico.core.dependencies.java.JavaDependency`

Special type of Java dependency for the Java libraries provided with Pimlico. These are packages up in jars and stored in the build dir.

check_java_dependency(*class_name*, *classpath=None*)

Utility to check that a java class is able to be loaded.

check_java()

Check that the JVM executable can be found. Raises a DependencyError if it can't be found or can't be run.

get_classpath(*deps*, *as_list=False*)

Given a list of JavaDependency subclass instances, returns all the components of the classpath that will make sure that the dependencies are available.

If as_list=True, returned as a list. Get the full classpath by ":".join(x) on the list. If as_list=False, returns classpath string.

get_module_classpath(*module*)

Builds a classpath that includes all of the classpath elements specified by Java dependencies of the given module. These include the dependencies from get_software_dependencies() and also any dependencies of the datatype.

Used to ensure that Java modules that depend on particular jars or classes get all of those files included on their classpath when Java is run.

class Py4JSoftwareDependency

Bases: `pimlico.core.dependencies.java.JavaDependency`

Java component of Py4J. Use this one as the main dependency, as it depends on the Python component and will install that first if necessary.

dependencies()

Returns a list of instances of :class:SoftwareDependency subcalsses representing this library's own dependencies. If the library is already available, these will never be consulted, but if it is to be installed, we will check first that all of these are available (and try to install them if not).

jars

installable()

Return True if it's possible to install this library automatically. If False, the user will have to install it themselves. Instructions for doing this may be provided by installation_instructions(), which will only generally be called if installable() returns False.

This might be the case, for example, if the software is not available to download freely, or if it requires a system-wide installation.

install(*local_config*, *trust_downloaded_archives=False*)

Should be overridden by any subclasses whose library is automatically installable. Carries out the actual installation.

You may assume that all dependencies returned by :method:dependencies have been satisfied prior to calling this.

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*, ***kwargs*)

Bases: `pimlico.core.dependencies.python.PythonPackageDependency`

Python package that can be installed via pip. Will be installed in the virtualenv if not available.

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.

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.

pimlico.core.dependencies.versions module

class SoftwareVersion(*string_id*)

Bases: `object`

Base class for representing version numbers / IDs of software. Different software may use different conventions to represent its versions, so it may be necessary to subclass this class to provide the appropriate parsing and comparison of versions.

compare_dotted_versions(*version0*, *version1*)

Comparison function for reasonably standard version numbers, with subversions to any level of nesting specified by dots.

Module contents

pimlico.core.external package

Submodules

pimlico.core.external.java module

call_java(*class_name*, *args=[]*, *classpath=None*)

start_java_process(*class_name*, *args=[]*, *java_args=[]*, *wait=0.1*, *classpath=None*)

class Py4JInterface(*gateway_class*, *port=None*, *python_port=None*, *gateway_args=[]*, *pipeline=None*, *print_stdout=True*, *print_stderr=True*, *env={}*, *system_properties={}*, *java_opts=[]*, *timeout=10.0*, *prefix_classpath=None*)

Bases: `object`

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

class DocumentMapOutputTypeWrapper(**args*, ***kwargs*)
Bases: `object`

non_filter_datatype = None

wrapped_module_info = None

output_name = None

archive_iter(*subsample=None*, *start_after=None*)
Provides an iterator just like TarredCorpus, but instead of iterating over data read from disk, gets it on the fly from the input datatype.

data_ready()
Ready to supply this data as soon as all the wrapper module's inputs are ready to produce their data.

wrap_module_info_as_filter(*module_info_instance*)
Create a filter module from a document map module so that it gets executed on the fly to provide its outputs as input to later modules. Can be applied to any document map module simply by adding *filter=T* to its config.

This function is called when *filter=T* is given.

Parameters **module_info_instance** – basic module info to wrap the outputs of

Returns a new non-executable ModuleInfo whose outputs are produced on the fly and will be identical to the outputs of the wrapper module.

pimlico.core.modules.map.multiproc module

Document map modules can in general be easily parallelized using multiprocessing. This module provides implementations of a pool and base worker processes that use multiprocessing, making it dead easy to implement a parallelized module, simply by defining what should be done on each document.

In particular, use :fun:.multiprocessing_executor_factory wherever possible.

class MultiprocessingMapProcess(*input_queue*, *output_queue*, *exception_queue*, *executor*, *docs_per_batch=1*)

Bases: `multiprocessing.process.Process`, `pimlico.core.modules.map.DocumentMapProcessMixin`

A base implementation of document map parallelization using multiprocessing. Note that not all document map modules will want to use this: e.g. if you call a background service that provides parallelization itself (like the CoreNLP module) there's no need for multiprocessing in the Python code.

notify_no_more_inputs()

Called when there aren't any more inputs to come.

run()

Method to be run in sub-process; can be overridden in sub-class

class MultiprocessingMapPool(*executor*, *processes*)

Bases: `pimlico.core.modules.map.DocumentProcessorPool`

A base implementation of document map parallelization using multiprocessing.

PROCESS_TYPE = None

SINGLE_PROCESS_TYPE = None

start_worker()

static create_queue(*maxsize=None*)

shutdown()

notify_no_more_inputs()

empty_all_queues()

class MultiprocessingMapModuleExecutor(*module_instance_info*, ***kwargs*)

Bases: `pimlico.core.modules.map.DocumentMapModuleExecutor`

POOL_TYPE = None

create_pool(*processes*)

Should return an instance of the pool to be used for document processing. Should generally be a subclass of DocumentProcessorPool.

Always called after preprocess().

postprocess(*error=False*)

Allows subclasses to define a finishing procedure to be called after corpus processing if finished.

multiprocessing_executor_factory(*process_document_fn*, *preprocess_fn=None*, *postprocess_fn=None*, *worker_set_up_fn=None*, *worker_tear_down_fn=None*, *batch_docs=None*, *multiprocessing_single_process=False*)

Factory function for creating an executor that uses the multiprocessing-based implementations of document-map pools and worker processes. This is an easy way to implement a parallelizable executor, which is suitable for a large number of module types.

process_document_fn should be a function that takes the following arguments (unless *batch_docs* is given):

- the worker process instance (allowing access to things set during setup)
- archive name
- document name
- the rest of the args are the document itself, from each of the input corpora

If proprocess_fn is given, it is called from the main process once before execution begins, with the executor as an argument.

If postprocess_fn is given, it is called from the main process at the end of execution, including on the way out after an error, with the executor as an argument and a kwarg *error* which is True if execution failed.

If worker_set_up_fn is given, it is called within each worker before execution begins, with the worker process instance as an argument. Likewise, worker_tear_down_fn is called from within the worker process before it exits.

Alternatively, you can supply a worker type, a subclass of :class:.MultiprocessingMapProcess, as the first argument. If you do this, worker_set_up_fn and worker_tear_down_fn will be ignored.

If *batch_docs* is not None, *process_document_fn* is treated differently. Instead of supplying the *process_document()* of the worker, it supplies a *process_documents()*. The second argument is a list of tuples, each of which is assumed to be the args to *process_document()* for a single document. In this case, *docs_per_batch* is set on the worker processes, so that the given number of docs are collected from the input and passed into *process_documents()* at once.

By default, if only a single process is needed, we use the threaded implementation of a map process instead of multiprocessing. If this doesn't work out in your case, for some reason, specify *multiprocessing_single_process=True* and a mutiprocessing process will be used even when only creating one.

pimlico.core.modules.map.singleproc module

Sometimes the simple multiprocessing-based approach to map module parallelization just isn't suitable. This module provides an equivalent set of implementations and convenience functions that don't use multiprocessing, but conform to the pool-based execution pattern by creating a single-thread pool.

class SingleThreadMapModuleExecutor(*module_instance_info*, ***kwargs*)

Bases: `pimlico.core.modules.map.threaded.ThreadingMapModuleExecutor`

create_pool(*processes*)

Should return an instance of the pool to be used for document processing. Should generally be a subclass of DocumentProcessorPool.

Always called after preprocess().

single_process_executor_factory(*process_document_fn*, *preprocess_fn=None*, *postprocess_fn=None*, *worker_set_up_fn=None*, *worker_tear_down_fn=None*, *batch_docs=None*)

Factory function for creating an executor that uses the single-process implementations of document-map pools and workers. This is an easy way to implement a non-parallelized executor

process_document_fn should be a function that takes the following arguments:

- the executor instance (allowing access to things set during setup)
- archive name
- document name
- the rest of the args are the document itself, from each of the input corpora

If proprocess_fn is given, it is called once before execution begins, with the executor as an argument.

If postprocess_fn is given, it is called at the end of execution, including on the way out after an error, with the executor as an argument and a kwarg *error* which is True if execution failed.

pimlico.core.modules.map.threaded module

Just like multiprocessing, but using threading instead. If you're not sure which you should use, it's probably multiprocessing.

class ThreadingMapThread(*input_queue*, *output_queue*, *exception_queue*, *executor*)
Bases: `threading.Thread`, *pimlico.core.modules.map.DocumentMapProcessMixin*

notify_no_more_inputs()
Called when there aren't any more inputs to come.

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.

shutdown(*timeout=3.0*)

class ThreadingMapPool(*executor*, *processes*)
Bases: *pimlico.core.modules.map.DocumentProcessorPool*

THREAD_TYPE = None

start_worker()

static create_queue(*maxsize=None*)

shutdown()

class ThreadingMapModuleExecutor(*module_instance_info*, ***kwargs*)
Bases: *pimlico.core.modules.map.DocumentMapModuleExecutor*

POOL_TYPE = None

create_pool(*processes*)
Should return an instance of the pool to be used for document processing. Should generally be a subclass of DocumentProcessorPool.

Always called after preprocess().

postprocess(*error=False*)
Allows subclasses to define a finishing procedure to be called after corpus processing if finished.

threading_executor_factory(*process_document_fn*, *preprocess_fn=None*, *postprocess_fn=None*, *worker_set_up_fn=None*, *worker_tear_down_fn=None*)
Factory function for creating an executor that uses the threading-based implementations of document-map pools and worker processes.

process_document_fn should be a function that takes the following arguments:

- the worker process instance (allowing access to things set during setup)
- archive name
- document name
- the rest of the args are the document itself, from each of the input corpora

If proprocess_fn is given, it is called from the main thread once before execution begins, with the executor as an argument.

If postprocess_fn is given, it is called from the main thread at the end of execution, including on the way out after an error, with the executor as an argument and a kwarg *error* which is True if execution failed.

If worker_set_up_fn is given, it is called within each worker before execution begins, with the worker thread instance as an argument. Likewise, worker_tear_down_fn is called from within the worker thread before it exits.

Alternatively, you can supply a worker type, a subclass of :class:.ThreadingMapThread, as the first argument. If you do this, worker_set_up_fn and worker_tear_down_fn will be ignored.

Module contents

class DocumentMapModuleInfo(*module_name*, *pipeline*, ***kwargs*)

Bases: `pimlico.core.modules.base.BaseModuleInfo`

Abstract module type that maps each document in turn in a corpus. It produces a single output document for every input.

Subclasses should specify the input types, which should all be subclasses of TarredCorpus, and output types, the first of which (i.e. default) should also be a subclass of TarredCorpus. The base class deals with iterating over the input(s) and writing the outputs to a new TarredCorpus. The subclass only needs to implement the mapping function applied to each document (in its executor).

module_outputs = [('documents', <class 'pimlico.datatypes.tar.TarredCorpus'>)]

input_corpora

get_writer(*output_name*, *output_dir*, *append=False*)

Get the writer instance that will be given processed documents to write. Should return a subclass of TarredCorpusWriter. The default implementation instantiates a plain TarredCorpusWriter.

get_writers(*append=False*)

get_detailed_status()

Returns a list of strings, containing detailed information about the module's status that is specific to the module type. This may include module-specific information about execution status, for example.

Subclasses may override this to supply useful (human-readable) information specific to the module type. They should called the super method.

class DocumentMapModuleExecutor(*module_instance_info*, ***kwargs*)

Bases: `pimlico.core.modules.base.BaseModuleExecutor`

Base class for executors for document map modules. Subclasses should provide the behaviour for each individual document by defining a pool (and worker processes) to handle the documents as they're fed into it.

Note that in most cases it won't be necessary to override the pool and worker base classes yourself. Unless you need special behaviour, use the standard implementations and factory functions.

Although the pattern of execution for all document map modules is based on parallel processing (creating a pool, spawning worker processes, etc), this doesn't mean that all such modules have to be parallelizable. If you have no reason not to parallelize, it's recommended that you do (with single-process execution as a special case). However, sometimes parallelizing isn't so simple: in these cases, consider using the tools in :mod:.singleproc.

preprocess()

Allows subclasses to define a set-up procedure to be called before corpus processing begins.

postprocess(*error=False*)

Allows subclasses to define a finishing procedure to be called after corpus processing if finished.

create_pool(*processes*)

Should return an instance of the pool to be used for document processing. Should generally be a subclass of DocumentProcessorPool.

Always called after preprocess().

retrieve_processing_status()

update_processing_status(*docs_completed*, *archive_name*, *filename*)

execute()
Run the actual module execution.

May return None, in which case it's assumed to have fully completed. If a string is returned, it's used as an alternative module execution status. Used, e.g., by multi-stage modules that need to be run multiple times.

skip_invalid(*fn*)
Decorator to apply to document map executor process_document() methods where you want to skip doing any processing if any of the input documents are invalid and just pass through the error information.

Be careful not to confuse this with the process_document() methods on datatypes. You don't need a decorator on them to skip invalid documents, as it's not called on them anyway.

skip_invalids(*fn*)
Decorator to apply to document map executor process_documents() methods where you want to skip doing any processing if any of the input documents are invalid and just pass through the error information.

invalid_doc_on_error(*fn*)
Decorator to apply to process_document() methods that causes all exceptions to be caught and an InvalidDocument to be returned as the result, instead of letting the error propagate up and call a halt to the whole corpus processing.

invalid_docs_on_error(*fn*)
Decorator to apply to process_documents() methods that causes all exceptions to be caught and an InvalidDocument to be returned as the result for every input document.

class ProcessOutput(*archive*, *filename*, *data*)
Bases: `object`

Wrapper for all result data coming out from a worker.

class InputQueueFeeder(*input_queue*, *iterator*, *complete_callback=None*)
Bases: `threading.Thread`

Background thread to read input documents from an iterator and feed them onto an input queue for worker processes/threads.

get_next_output_document()

check_invalid(*archive*, *filename*)
Checks whether a given document was invalid in the input. Once the check has been performed, the item is removed from the list, for efficiency, so this should only be called once per document.

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.

check_for_error()
Can be called from the main thread to check whether an error has occurred in this thread and raise a suitable exception if so

shutdown(*timeout=3.0*)
Cancel the feeder, if it's still feeding and stop the thread. Call only after you're sure you no longer need anything from any of the queues. Waits for the thread to end.

Call from the main thread (that created the feeder) only.

class DocumentProcessorPool(*processes*)
Bases: `object`

Base class for pools that provide an easy implementation of parallelization for document map modules. Defines the core interface for pools.

If you're using multiprocessing, you'll want to use the multiprocessing-specific subclass.

notify_no_more_inputs()

static create_queue(*maxsize=None*)
May be overridden by subclasses to provide different implementations of a Queue. By default, uses the multiprocessing queue type. Whatever is returned, it should implement the interface of Queue.Queue.

shutdown()

empty_all_queues()

class DocumentMapProcessMixin(*input_queue*, *output_queue*, *exception_queue*, *docs_per_batch=1*)
Bases: `object`

Mixin/base class that should be implemented by all worker processes for document map pools.

set_up()
Called when the process starts, before it starts accepting documents.

process_document(*archive*, *filename*, **docs*)

process_documents(*doc_tuples*)
Batched version of process_document(). Default implementation just calls process_document() on each document, but if you want to group documents together and process multiple at once, you can override this method and make sure the *docs_per_batch* is set > 1.

Each item in the list of doc tuples should be a tuple of the positional args to process_document() – i.e. archive_name, filename, doc_from_corpus1, [doc_from corpus2, . . .]

tear_down()
Called from within the process after processing is complete, before exiting.

notify_no_more_inputs()
Called when there aren't any more inputs to come.

exception WorkerStartupError(**args*, ***kwargs*)
Bases: `exceptions.Exception`

exception WorkerShutdownError(**args*, ***kwargs*)
Bases: `exceptions.Exception`

Submodules

pimlico.core.modules.base module

This module provides base classes for Pimlico modules.

The procedure for creating a new module is the same whether you're contributing a module to the core set in the Pimlico codebase or a standalone module in your own codebase, or for a specific pipeline.

A Pimlico module is identified by the full Python-path to the Python package that contains it. This package should be laid out as follows:

- The module's metadata is defined by a class in info.py called ModuleInfo, which should inherit from BaseModuleInfo or one of its subclasses.
- The module's functionality is provided by a class in execute.py called ModuleExecutor, which should inherit from BaseModuleExecutor.

The exec Python module will not be imported until an instance of the module is to be run. This means that you can import dependencies and do any necessary initialization at the point where it's executed, without worrying about incurring the associated costs (and dependencies) every time a pipeline using the module is loaded.

class BaseModuleInfo(*module_name*, *pipeline*, *inputs={}*, *options={}*, *optional_outputs=[]*, *docstring="*, *include_outputs=[]*, *alt_expanded_from=None*, *alt_param_settings=[]*, *module_variables={}*)

Bases: `object`

Abstract base class for all pipeline modules' metadata.

module_type_name = None

module_readable_name = None

module_options = {}

Specifies a list of (name, datatype class) pairs for inputs that are always required

module_inputs = []

Specifies a list of (name, datatype class) pairs for optional inputs. The module's execution may vary depending on what is provided. If these are not given, None is returned from get_input()

module_optional_inputs = []

Specifies a list of (name, datatype class) pairs for outputs that are always written

module_optional_outputs = []

Whether the module should be executed Typically True for almost all modules, except input modules (though some of them may also require execution) and filters

module_executable = True

If specified, this ModuleExecutor class will be used instead of looking one up in the exec Python module

module_executor_override = None

Usually None. In the case of stages of a multi-stage module, stores a pointer to the main module.

main_module = None

module_outputs = []

Specifies a list of (name, datatype class) pairs for outputs that are written only if they're specified in the "output" option or used by another module

load_executor()

Loads a ModuleExecutor for this Pimlico module. Usually, this just involves calling *load_module_executor()*, but the default executor loading may be overridden for a particular module type by overriding this function. It should always return a subclass of ModuleExecutor, unless there's an error.

classmethod get_key_info_table()

When generating module docs, the table at the top of the page is produced by calling this method. It should return a list of two-item lists (title + value). Make sure to include the super-class call if you override this to add in extra module-specific info.

metadata_filename

get_metadata()

set_metadata_value(*attr*, *val*)

set_metadata_values(*val_dict*)

status

execution_history_path

add_execution_history_record(*line*)
Output a single line to the file that stores the history of module execution, so we can trace what we've done.

execution_history
Get the entire recorded execution history for this module. Returns an empty string if no history has been recorded.

input_names
All required inputs, first, then all supplied optional inputs

output_names

classmethod process_module_options(*opt_dict*)
Parse the options in a dictionary (probably from a config file), checking that they're valid for this model type.

Parameters `opt_dict` – dict of options, keyed by option name

Returns dict of options

classmethod extract_input_options(*opt_dict*, *module_name=None*, *previous_module_name=None*, *module_expansions={}*)
Given the config options for a module instance, pull out the ones that specify where the inputs come from and match them up with the appropriate input names.

The inputs returned are just names as they come from the config file. They are split into module name and output name, but they are not in any way matched up with the modules they connect to or type checked.

Parameters

- `module_name` – name of the module being processed, for error output. If not given, the name isn't included in the error.
- `previous_module_name` – name of the previous module in the order given in the config file, allowing a single-input module to default to connecting to this if the input connection wasn't given
- `module_expansions` – dictionary mapping module names to a list of expanded module names, where expansion has been performed as a result of alternatives in the parameters. Provided here so that the unexpanded names may be used to refer to the whole list of module names, where a module takes multiple inputs on one input parameter

Returns dictionary of inputs

static get_extra_outputs_from_options(*options*)
Normally, which optional outputs get produced by a module depend on the 'output' option given in the config file, plus any outputs that get used by subsequent modules. By overriding this method, module types can add extra outputs into the list of those to be included, conditional on other options.

It also receives the processed dictionary of inputs, so that the additional outputs can depend on what is fed into the input.

E.g. the corenlp module include the 'annotations' output if annotators are specified, so that the user doesn't need to give both options.

`provide_further_outputs`()
Called during instantiation, once inputs and options are available, to add a further list of module outputs that are dependent on inputs or options.

`get_module_output_dir`(*short_term_store=False*)
Gets the path to the base output dir to be used by this module, relative to the storage base dir. When outputting data, the storage base dir will always be the short term store path, but when looking for the output data other base paths might be explored, including the long term store.

Parameters **`short_term_store`** – if True, return absolute path to output dir in short-term store (used for output)

Returns path, relative to store base path, or if short_term_store=True absolute path to output dir

`get_absolute_output_dir`(*output_name*)
The simplest way to get hold of the directory to use to output data to for a given output. This is the usual way to get an output directory for an output writer.

The directory is an absolute path to a location in the Pimlico short-term store.

Parameters **`output_name`** – the name of an output

Returns the absolute path to the output directory to use for the named output

`get_output_dir`(*output_name*, *short_term_store=False*)

Parameters

- **`short_term_store`** – return an absolute path in the short-term store. If False (default), return a relative path, specified relative to the root of the Pimlico store used. This allows multiple stores to be searched for output
- **`output_name`** – the name of an output

Returns the path to the output directory to use for the named output, which may be relative to the root of the Pimlico store in use (default) or an absolute path in the short-term store, depending on *short_term_store*

`get_output_datatype`(*output_name=None*, *additional_names=[]*)

`instantiate_output_datatype`(*output_name*, *output_datatype*, ***kwargs*)
Subclasses may want to override this to provide special behaviour for instantiating particular outputs' datatypes.

Additional kwargs will be pass through to the datatype's init.

`get_output`(*output_name=None*, *additional_names=None*, ***kwargs*)
Get a datatype instance corresponding to one of the outputs of the module.

Additional kwargs will be pass through to the datatype's init.

`is_multiple_input`(*input_name=None*)
Returns True if the named input (or default input if no name is given) is a MultipleInputs input, False otherwise. If it is, get_input() will return a list, otherwise it will return a single datatype.

`get_input_module_connection`(*input_name=None*, *always_list=False*)
Get the ModuleInfo instance and output name for the output that connects up with a named input (or the first input) on this module instance. Used by get_input() – most of the time you probably want to use that to get the instantiated datatype for an input.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single (module, output_name) pair. If always_list=True, in this latter case we return a single-item list.

`get_input_datatype`(*input_name=None*, *always_list=False*)
Get a list of datatype classes corresponding to one of the inputs to the module. If an input name is not given, the first input is returned.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single datatype.

`get_input`(*input_name=None*, *always_list=False*, ***kwargs*)
Get a datatype instances corresponding to one of the inputs to the module. Looks up the corresponding output from another module and uses that module's metadata to get that output's instance. If an input name is not given, the first input is returned.

If the input type was specified with MultipleInputs, meaning that we're expecting an unbounded number of inputs, this is a list. Otherwise, it's a single datatype instance. If always_list=True, in this latter case we return a single-item list.

If the requested input name is an optional input and it has not been supplied, returns None.

Additional kwargs will be passed through to the datatype's init call.

`input_ready`(*input_name=None*)
Check whether the datatype is (or datatypes are) ready to go, corresponding to the named input.

Parameters **`input_name`** – input to check

Returns True if input is ready

`all_inputs_ready`()
Check *input_ready()* on all inputs.

Returns True if all input datatypes are ready to be used

`classmethod is_filter`()

`missing_module_data`()
Reports missing data not associated with an input dataset.

Calling *missing_data()* reports any problems with input data associated with a particular input to this module. However, modules may also rely on data that does not come from one of their inputs. This happens primarily (perhaps solely) when a module option points to a data source. This might be the case with any module, but is particularly common among input reader modules, which have no inputs, but read data according to their options.

Returns list of problems

`missing_data`(*input_names=None*, *assume_executed=[]*, *assume_failed=[]*, *allow_preliminary=False*)
Check whether all the input data for this module is available. If not, return a list strings indicating which outputs of which modules are not available. If it's all ready, returns an empty list.

To check specific inputs, give a list of input names. To check all inputs, don't specify *input_names*. To check the default input, give *input_names=[None]*. If not checking a specific input, also checks non-input data (see *missing_module_data()*).

If *assume_executed* is given, it should be a list of module names which may be assumed to have been executed at the point when this module is executed. Any outputs from those modules will be excluded from the input checks for this module, on the assumption that they will have become available, even if they're not currently available, by the time they're needed.

If *assume_executed* is given, it should be a list of module names which should be assumed to have failed. If we rely on data from the output of one of them, instead of checking whether it's available we simply assume it's not.

Why do this? When running multiple modules in sequence, if one fails it is possible that its output datasets look like complete datasets. For example, a partially written iterable corpus may look like a perfectly valid corpus, which happens to be smaller than it should be. After the execution failure, we may check other modules to see whether it's possible to run them. Then we need to know not to trust the output data from the failed module, even if it looks valid.

If *allow_preliminary=True*, for any inputs that are multiple inputs and have multiple connections to previous modules, consider them to be satisfied if at least one of their inputs is ready. The normal behaviour is to require all of them to be ready, but in a preliminary run this requirement is relaxed.

classmethod is_input()

dependencies

Returns list of names of modules that this one depends on for its inputs.

get_transitive_dependencies()

Transitive closure of *dependencies*.

Returns list of names of modules that this one recursively (transitively) depends on for its inputs.

get_input_type_requirements(*input_name=None*)

typecheck_inputs()

typecheck_input(*input_name*)

Typecheck a single input. `typecheck_inputs()` calls this and is used for typechecking of a pipeline. This method returns the (or the first) satisfied input requirement, or raises an exception if typechecking failed, so can be handy separately to establish which requirement was met.

The result is always a list, but will contain only one item unless the input is a multiple input.

get_software_dependencies()

Check that all software required to execute this module is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

get_input_software_dependencies()

Collects library dependencies from the input datatypes to this module, which will need to be satisfied for the module to be run.

Unlike *`get_software_dependencies()`*, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

get_output_software_dependencies()

Collects library dependencies from the output datatypes to this module, which will need to be satisfied for the module to be run.

Unlike *`get_input_software_dependencies()`*, it may not be the case that all of these dependencies strictly need to be satisfied before the module can be run. It could be that a datatype can be written without satisfying all the dependencies needed to read it. However, we assume that dependencies of all output datatypes must be satisfied in order to run the module that writes them, since this is usually the case, and these are checked before running the module.

Unlike `get_software_dependencies()`, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

`check_ready_to_run``()`
Called before a module is run, or if the 'check' command is called. This will only be called after all library dependencies have been confirmed ready (see :method:get_software_dependencies).

Essentially, this covers any module-specific checks that used to be in check_runtime_dependencies() other than library installation (e.g. checking models exist).

Always call the super class' method if you override.

Returns a list of (name, description) pairs, where the name identifies the problem briefly and the description explains what's missing and (ideally) how to fix it.

`reset_execution``()`
Remove all output data and metadata from this module to make a fresh start, as if it's never been executed.

May be overridden if a module has some side effect other than creating/modifying things in its output directory(/ies), but overridden methods should always call the super method. Occasionally this is necessary, but most of the time the base implementation is enough.

`get_detailed_status``()`
Returns a list of strings, containing detailed information about the module's status that is specific to the module type. This may include module-specific information about execution status, for example.

Subclasses may override this to supply useful (human-readable) information specific to the module type. They should called the super method.

`classmethod module_package_name``()`
The package name for the module, which is used to identify it in config files. This is the package containing the info.py in which the ModuleInfo is defined.

`get_execution_dependency_tree``()`
Tree of modules that will be executed when this one is executed. Where this module depends on filters, the tree goes back through them to find what they depend on (since they will be executed simultaneously)

`get_all_executed_modules``()`
Returns a list of all the modules that will be executed when this one is (including itself). This is the current module (if executable), plus any filters used to produce its inputs.

`lock_path`

`lock``()`
Mark the module as locked, so that it cannot be executed. Called when execution begins, to ensure that you don't end up executing the same module twice simultaneously.

`unlock``()`
Remove the execution lock on this module.

`is_locked``()`

Returns True is the module is currently locked from execution

`get_new_log_filename``(`*name='error'*`)`
Returns an absolute path that can be used to output a log file for this module. This is used for outputting error logs. It will always return a filename that doesn't currently exist, so can be used multiple times to output multiple logs.

`collect_unexecuted_dependencies``(`*modules*`)`
Given a list of modules, checks through all the modules that they depend on to put together a list of modules that need to be executed so that the given list will be left in an executed state. The list includes the modules themselves, if they're not fully executed, and unexecuted dependencies of any unexecuted modules (recursively).

Parameters **`modules`** – list of ModuleInfo instances

Returns list of ModuleInfo instances that need to be executed

`collect_runnable_modules`(*pipeline*, *preliminary=False*)
Look for all unexecuted modules in the pipeline to find any that are ready to be executed. Keep collecting runnable modules, including those that will become runnable once we've run earlier ones in the list, to produce a list of a sequence of modules that could be set running now.

Parameters **`pipeline`** – pipeline config

Returns ordered list of runable modules. Note that it must be run in this order, as some might depend on earlier ones in the list

`satisfies_typecheck`(*provided_type*, *type_requirements*)
Interface to Pimlico's standard type checking (see *check_type*) that returns a boolean to say whether type checking succeeded or not.

`check_type`(*provided_type*, *type_requirements*)
Type-checking algorithm for making sure outputs from modules connect up with inputs that they satisfy the requirements for.

`class BaseModuleExecutor`(*module_instance_info*, *stage=None*, *debug=False*, *force_rerun=False*)
Bases: `object`

Abstract base class for executors for Pimlico modules. These are classes that actually do the work of executing the module on given inputs, writing to given output locations.

`execute`()
Run the actual module execution.

May return None, in which case it's assumed to have fully completed. If a string is returned, it's used as an alternative module execution status. Used, e.g., by multi-stage modules that need to be run multiple times.

`exception ModuleInfoLoadError`(**args*, ***kwargs*)
Bases: `exceptions.Exception`

`exception ModuleExecutorLoadError`
Bases: `exceptions.Exception`

`exception ModuleTypeError`
Bases: `exceptions.Exception`

`exception TypeCheckError`
Bases: `exceptions.Exception`

`exception DependencyError`(*message*, *stderr=None*, *stdout=None*)
Bases: `exceptions.Exception`

Raised when a module's dependencies are not satisfied. Generally, this means a dependency library needs to be installed, either on the local system or (more often) by calling the appropriate make target in the lib directory.

`load_module_executor`(*path_or_info*)
Utility for loading the executor class for a module from its full path. More or less just a wrapper around an import, with some error checking. Locates the executor by a standard procedure that involves checking for an "execute" python module alongside the info's module.

Note that you shouldn't generally use this directly, but instead call the *load_executor()* method on a module info (which will call this, unless special behaviour has been defined).

Parameters **`path`** – path to Python package containing the module

Returns class

load_module_info(*path*)

Utility to load the metadata for a Pimlico pipeline module from its package Python path.

Parameters **path** –

Returns

pimlico.core.modules.execute module

Runtime execution of modules

This module provides the functionality to check that Pimlico modules are ready to execute and execute them. It is used by the *run* command.

check_and_execute_modules(*pipeline*, *module_names*, *force_rerun=False*, *debug=False*, *log=None*, *all_deps=False*, *check_only=False*, *exit_on_error=False*, *preliminary=False*, *email=None*)

Main method called by the *run* command that first checks a pipeline, checks all pre-execution requirements of the modules to be executed and then executes each of them. The most common case is to execute just one module, but a sequence may be given.

Parameters

- **exit_on_error** – drop out if a ModuleExecutionError occurs in any individual module, instead of continuing to the next module that can be run
- **pipeline** – loaded PipelineConfig
- **module_names** – list of names of modules to execute in the order they should be run
- **force_rerun** – execute modules, even if they're already marked as complete
- **debug** – output debugging info
- **log** – logger, if you have one you want to reuse
- **all_deps** – also include unexecuted dependencies of the given modules
- **check_only** – run all checks, but stop before executing. Used for *check* command

Returns

check_modules_ready(*pipeline*, *modules*, *log*, *preliminary=False*)

Check that a module is ready to be executed. Always called before execution begins.

Parameters

- **pipeline** – loaded PipelineConfig
- **modules** – loaded ModuleInfo instances, given in the order they're going to be executed. For each module, it's assumed that those before it in the list have already been run when it is run.
- **log** – logger to output to

Returns If *preliminary=True*, list of problems that were ignored by allowing preliminary run. Otherwise, None – we raise an exception when we first encounter a problem

execute_modules(*pipeline*, *modules*, *log*, *force_rerun=False*, *debug=False*, *exit_on_error=False*, *preliminary=False*, *email=None*)

format_execution_dependency_tree(*tree*)

send_final_report_email(*pipeline*, *error_modules*, *success_modules*, *skipped_modules*, *all_modules*)

send_module_report_email(*pipeline*, *module*, *short_error*, *long_error*)

exception ModuleExecutionError(**args*, ***kwargs*)
Bases: `exceptions.Exception`

exception ModuleNotReadyError(**args*, ***kwargs*)
Bases: `pimlico.core.modules.execute.ModuleExecutionError`

exception ModuleAlreadyCompletedError(**args*, ***kwargs*)
Bases: `pimlico.core.modules.execute.ModuleExecutionError`

exception StopProcessing
Bases: `exceptions.Exception`

pimlico.core.modules.inputs module

Base classes and utilities for input modules in a pipeline.

class InputModuleInfo(*module_name*, *pipeline*, *inputs={}*, *options={}*, *optional_outputs=[]*, *docstring=''*, *include_outputs=[]*, *alt_expanded_from=None*, *alt_param_settings=[]*, *module_variables={}*)
Bases: `pimlico.core.modules.base.BaseModuleInfo`

Base class for input modules. These don't get executed in general, they just provide a way to iterate over input data.

You probably don't want to subclass this. It's usually simplest to define a datatype for reading the input data and then just specify its class as the module's type. This results in a subclass of this module info being created dynamically to read that data.

Note that module_executable is typically set to False and the base class does this. However, some input modules need to be executed before the input is usable, for example to collect stats about the input data.

module_type_name = 'input'

module_executable = False

instantiate_output_datatype(*output_name*, *output_datatype*, ***kwargs*)
Subclasses may want to override this to provide special behaviour for instantiating particular outputs' datatypes.

Additional kwargs will be pass through to the datatype's init.

input_module_factory(*datatype*)
Create an input module class to load a given datatype.

class ReaderOutputType(*reader_options*, *base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.base.IterableCorpus`

A datatype for reading in input according to input module options and allowing it to be iterated over by other modules.

Typically used together with *iterable_input_reader_factory()* as the output datatype.

`__len__` should be overridden to take the processed input module options and return the length of the corpus (number of documents).

`__iter__` should use the processed input module options and return an iterator over the corpus' documents (e.g. a generator function). Each item yielded should be a pair `(doc_name, data)` and `data` should be in the appropriate internal format associated with the document type.

`data_ready` should be overridden to use the processed input module options and return True if the data is ready to be read in.

In all cases, the input options are available as `self.reader_options`.

`datatype_name = 'reader_iterator'`

`data_point_type = None`
Must be overridden by subclasses

`emulated_datatype`
alias of `pimlico.datatypes.base.IterableCorpus`

`data_ready``()`
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

`class DocumentCounterModuleExecutor``(`*module_instance_info*, *stage=None*, *debug=False*, *force_rerun=False*`)`
Bases: `pimlico.core.modules.base.BaseModuleExecutor`

An executor that just calls the __len__ method to count documents and stores the result

`execute``()`
Run the actual module execution.

May return None, in which case it's assumed to have fully completed. If a string is returned, it's used as an alternative module execution status. Used, e.g., by multi-stage modules that need to be run multiple times.

`decorate_require_stored_len``(`*obj*`)`
Decorator for a data_ready() function that requires the data's length to have been computed. Used when execute_count==True.

`iterable_input_reader_factory``(`*input_module_options*, *output_type*, *module_type_name=None*, *module_readable_name=None*, *software_dependencies=None*, *execute_count=False*`)`
Factory for creating an input reader module type. This is a non-executable module that has no inputs. It reads its data from some external location, using the given module options. The resulting dataset is an IterableCorpus subtype, with the given document type.

`output_type` is a datatype that performs the actual iteration over the data and is instantiated with the processed options as its first argument. This is typically created by subclassing ReaderOutputType and providing len, iter and data_ready methods.

`software_dependencies` may be a list of software dependencies that the module-info will return when `get_software_dependencies()` is called, or a function that takes the module-info instance and returns such a list. If left blank, no dependencies are returned.

If `execute_count==True`, the module will be an executable module and the execution will simply count the number of documents in the corpus and store the count. This should be used if counting the documents in the dataset is not completely trivial and quick (e.g. if you need to read through the data itself, rather than something like counting files in a directory or checking metedata). It is common for this to be the only processing that needs to be done on the dataset before using it. The `output_type` should then implement a `count_documents()` method. The `__len__` method then simply use the computed and stored value. There is no need to override it.

If the `count_documents()` method returns a pair of integers, instead of just a single integer, they are taken to be the total number of documents in the corpus and the number of valid documents (i.e. the number that will be produce an InvalidDocument). In this case, the valid documents count is also stored in the metadata, as `valid_documents`.

How is this different from ``input_module_factory``? This method is used in your module code to prepare a ModuleInfo class for reading a particular type of input data and presenting it as a Pimlico dataset of the given type. `input_module_factory`, on the other hand, is used by Pimlico when you specify a datatype as a module type in a config file.

Note that, in future versions, reading datasets output by another Pimlico pipeline will be the only purpose for that special notation. The possibility of specifying `input_module_options` to create an input reader will disappear, so the use of `input_module_options` should be phased out and replaced with input reader modules, such as those created by this factory.

pimlico.core.modules.multistage module

class MultistageModuleInfo(*module_name*, *pipeline*, ***kwargs*)

Bases: `pimlico.core.modules.base.BaseModuleInfo`

Base class for multi-stage modules. You almost certainly don't want to override this yourself, but use the factory method instead. It exists mainly for providing a way of identifying multi-stage modules.

module_executable = True

stages = None

typecheck_inputs()

Overridden to check internal output-input connections as well as the main module's inputs.

get_software_dependencies()

Check that all software required to execute this module is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

get_input_software_dependencies()

Collects library dependencies from the input datatypes to this module, which will need to be satisfied for the module to be run.

Unlike `get_software_dependencies()`, it shouldn't need to be overridden by subclasses, since it just collects the results of getting dependencies from the datatypes.

check_ready_to_run()

Called before a module is run, or if the 'check' command is called. This will only be called after all library dependencies have been confirmed ready (see :method:get_software_dependencies).

Essentially, this covers any module-specific checks that used to be in check_runtime_dependencies() other than library installation (e.g. checking models exist).

Always call the super class' method if you override.

Returns a list of (name, description) pairs, where the name identifies the problem briefly and the description explains what's missing and (ideally) how to fix it.

get_detailed_status()

Returns a list of strings, containing detailed information about the module's status that is specific to the module type. This may include module-specific information about execution status, for example.

Subclasses may override this to supply useful (human-readable) information specific to the module type. They should called the super method.

reset_execution()

Remove all output data and metadata from this module to make a fresh start, as if it's never been executed.

May be overridden if a module has some side effect other than creating/modifying things in its output directory(/ies), but overridden methods should always call the super method. Occasionally this is necessary, but most of the time the base implementation is enough.

classmethod get_key_info_table()

Add the stages into the key info table.

get_next_stage()

If there are more stages to be executed, returns a pair of the module info and stage definition. Otherwise, returns (None, None)

status

is_locked()

Returns True is the module is currently locked from execution

multistage_module(*multistage_module_type_name*, *module_stages*, *use_stage_option_names=False*, *module_readable_name=None*)

Factory to build a multi-stage module type out of a series of stages, each of which specifies a module type for the stage. The stages should be a list of `ModuleStage` objects.

class ModuleStage(*name*, *module_info_cls*, *connections=None*, *output_connections=None*, *option_connections=None*, *use_stage_option_names=False*)

Bases: `object`

A single stage in a multi-stage module.

If no explicit input connections are given, the default input to this module is connected to the default output from the previous.

Connections can be given as a list of `ModuleConnection` s.

Output connections specify that one of this module's outputs should be used as an output from the multi-stage module. Optional outputs for the multi-stage module are not currently supported (though could in theory be added later). This should be a list of `ModuleOutputConnection` s. If none are given for any of the stages, the module will have a single output, which is the default output from the last stage.

Option connections allow you to specify the names that are used for the multistage module's options that get passed through to this stage's module options. Simply specify a dict for `option_connections` where the keys are names module options for this stage and the values are the names that should be used for the multistage module's options.

You may map multiple options from different stages to the same option name for the multistage module. This will result in the same option value being passed through to both stages. Note that help text, option type, option processing, etc will be taken from the first stage's option (in case the two options aren't identical).

Options not explicitly mapped to a name will use the name `<stage_name>_<option_name>`. If `use_stage_option_names=True`, this prefix will not be added: the stage's option names will be used directly as the option name of the multistage module. Note that there is a danger of clashing option names with this behaviour, so only do it if you know the stages have distinct option names (or should share their values where the names overlap).

class ModuleConnection

Bases: `object`

class InternalModuleConnection(*input_name*, *output_name=None*, *previous_module=None*)
Bases: `pimlico.core.modules.multistage.ModuleConnection`

Connection between the output of one module in the multi-stage module and the input to another.

May specify the name of the previous module that a connection should be made to. If this is not given, the previous module in the sequence will be assumed.

If *output_name=None*, connects to the default output of the previous module.

class ModuleInputConnection(*stage_input_name=None*, *main_input_name=None*)
Bases: `pimlico.core.modules.multistage.ModuleConnection`

Connection of a sub-module's input to an input to the multi-stage module.

If *main_input_name* is not given, the name for the input to the multistage module will be identical to the stage input name. This might lead to unintended behaviour if multiple inputs end up with the same name, so you can specify a different name if necessary to avoid clashes.

If multiple inputs (e.g. from different stages) are connected to the same main input name, they will take input from the same previous module output. Nothing clever is done to unify the type requirements, however: the first stage's type requirement is used for the main module's input.

If *stage_input_name* is not given, the module's default input will be connected.

class ModuleOutputConnection(*stage_output_name=None*, *main_output_name=None*)
Bases: `object`

Specifies the connection of a sub-module's output to the multi-stage module's output. Works in a similar way to *ModuleInputConnection*.

exception MultistageModulePreparationError
Bases: `exceptions.Exception`

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.

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=<type 'str'>*, *length=None*)
Option processor type that accepts comma-separated lists of strings. Each value is then parsed according to the given item_type (default: string).

comma_separated_strings(*string*)

json_string(*string*)

process_module_options(*opt_def*, *opt_dict*, *module_type_name*)
Utility for processing runtime module options. Called from module base class.

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.

pimlico.core.visualize package

Submodules

pimlico.core.visualize.deps module

class GraphvizDependency(***kwargs*)
Bases: *pimlico.core.dependencies.base.SystemCommandDependency*

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.

pimlico.core.visualize.status module

build_graph_with_status(*pipeline*, *all=False*)

Module contents

Visualization of pipelines using Graphviz.

This is not fully implemented yet. In fact, I've barely started. But you'll find some indication of where I'm going with it on the *Pimlico Wishlist*.

Note: Do not import anything from subpackages unless you're doing graph visualization, as they will trigger a check for Graphviz and try to install it.

Note: A bug in pygraphviz means that automatic installation on Ubuntu (and perhaps other systems) gets in a twist and leaves an unresolved dependency. If you have this problem and system-wide install is an option, just install with *sudo apt-get install python-pygraphviz*.

Submodules

pimlico.core.config module

Reading of pipeline config from a file into the data structure used to run and manipulate the pipeline's data.

class PipelineConfig(*name*, *pipeline_config*, *local_config*, *filename=None*, *variant='main'*, *available_variants=[]*, *log=None*, *all_filenames=None*, *module_aliases={}*, *local_config_sources=None*)

Bases: `object`

Main configuration for a pipeline, read in from a config file.

For details on how to write config files that get read by this class, see *Pipeline config*.

modules

List of module names, in the order they were specified in the config file.

module_dependencies

Dictionary mapping a module name to a list of the names of modules that it depends on for its inputs.

module_dependents

Opposite of module_dependencies. Returns a mapping from module names to a list of modules the depend on the module.

get_dependent_modules(*module_name*, *recurse=False*, *exclude=[]*)

Return a list of the names of modules that depend on the named module for their inputs.

If *exclude* is given, we don't perform a recursive call on any of the modules in the list. For each item we recurse on, we extend the exclude list in the recursive call to include everything found so far (in other recursive calls). This avoids unnecessary recursion in complex pipelines.

If *exclude=None*, it is also passed through to recursive calls as None. Its default value of *[]* avoids excessive recursion from the top-level call, by allowing things to be added to the exclusion list for recursive calls.

Parameters **recurse** – include all transitive dependents, not just those that immediately depend on the module.

append_module(*module_info*)

Add a moduleinfo to the end of the pipeline. This is mainly for use while loaded a pipeline from a config file.

get_module_schedule()

Work out the order in which modules should be executed. This is an ordering that respects dependencies, so that modules are executed after their dependencies, but otherwise follows the order in which modules were specified in the config.

Returns list of module names

reset_all_modules()

Resets the execution states of all modules, restoring the output dirs as if nothing's been run.

`path_relative_to_config`(*path*)
Get an absolute path to a file/directory that's been specified relative to a config file (usually within the config file).

Parameters **`path`** – relative path

Returns absolute path

`static load`(*filename*, *local_config=None*, *variant='main'*, *override_local_config={}*)
Main function that loads a pipeline from a config file.

Parameters

- **`filename`** – file to read config from
- **`local_config`** – location of local config file, where we'll read system-wide config. Usually not specified, in which case standard locations are searched. When loading programmatically, you might want to give this
- **`variant`** – pipeline variant to load
- **`override_local_config`** – extra configuration values to override the system-wide config

Returns

`static load_local_config`(*filename=None*, *override={}*)
Load local config parameters. These are usually specified in a *.pimlico* file, but may be overridden by other config locations, on the command line, or elsewhere programmatically.

`static empty`(*local_config=None*, *override_local_config={}*, *override_pipeline_config={}*)
Used to programmatically create an empty pipeline. It will contain no modules, but provides a gateway to system info, etc and can be used in place of a real Pimlico pipeline.

Parameters

- **`local_config`** – filename to load local config from. If not given, the default locations are searched
- **`override_local_config`** – manually override certain local config parameters. Dict of parameter values

Returns the *`PipelineConfig`* instance

`find_data_path`(*path*, *default=None*)
Given a path to a data dir/file relative to a data store, tries taking it relative to various store base dirs. If it exists in a store, that absolute path is returned. If it exists in no store, return None. If the path is already an absolute path, nothing is done to it.

The stores searched are the long-term store and the short-term store, though in the future more valid data storage locations may be added.

Parameters

- **`path`** – path to data, relative to store base
- **`default`** – usually, return None if no data is found. If default="short", return path relative to short-term store in this case. If default="long", long-term store.

Returns absolute path to data, or None if not found in any store

`find_all_data_paths`(*path*)

get_data_search_paths(*path*)
Like *find_all_data_paths()*, but returns a list of all absolute paths which this data path could correspond to, whether or not they exist.

Parameters **path** – relative path within Pimlico directory structures

Returns list of string

get_storage_roots()
Returns a list of all the (pipeline-specific) storage root locations known to the pipeline.

Currently, this is always *[self.short_term_store, self.long_term_store]*, but in future we may have a more flexible system that allows an unbounded number of storage locations.

step

enable_step()
Enable super-verbose, interactive step mode.

::seealso:

```
Module :mod:pimlico.cli.debug
   The debug module defines the behaviour of step mode.
```

exception PipelineConfigParseError(**args*, ***kwargs*)
Bases: `exceptions.Exception`

General problems interpreting pipeline config

exception PipelineStructureError
Bases: `exceptions.Exception`

Fundamental structural problems in a pipeline.

exception PipelineCheckError(*cause*, **args*, ***kwargs*)
Bases: `exceptions.Exception`

Error in the process of explicitly checking a pipeline for problems.

preprocess_config_file(*filename*, *variant='main'*, *initial_vars={}*)
Workhorse of the initial part of config file reading. Deals with all of our custom stuff for pipeline configs, such as preprocessing directives and includes.

Parameters

- **filename** – file from which to read main config
- **variant** – name of a variant to load. The default (*main*) loads the main variant, which always exists
- **initial_vars** – variable assignments to make available for substitution. This will be added to by any *vars* sections that are read.

Returns tuple: raw config dict; list of variants that could be loaded; final vars dict; list of filenames that were read, including included files; dict of docstrings for each config section

check_for_cycles(*pipeline*)
Basic cyclical dependency check, always run on pipeline before use.

check_release(*release_str*)
Check a release name against the current version of Pimlico to determine whether we meet the requirement.

check_pipeline(*pipeline*)
Checks a pipeline over for metadata errors, cycles, module typing errors and other problems. Called every time a pipeline is loaded, to check the whole pipeline's metadata is in order.

Raises a *PipelineCheckError* if anything's wrong.

get_dependencies(*pipeline*, *modules*, *recursive=False*, *sources=False*)
Get a list of software dependencies required by the subset of modules given.

If recursive=True, dependencies' dependencies are added to the list too.

Parameters

- **pipeline** –
- **modules** – list of modules to check. If None, checks all modules

print_missing_dependencies(*pipeline*, *modules*)
Check runtime dependencies for a subset of modules and output a table of missing dependencies.

Parameters

- **pipeline** –
- **modules** – list of modules to check. If None, checks all modules

Returns True if no missing dependencies, False otherwise

print_dependency_leaf_problems(*dep*, *local_config*)

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.coref package

Submodules

pimlico.datatypes.coref.corenlp module

Datatypes for coreference resolution output. Based on Stanford CoreNLP's coref output, so includes all the information provided by that.

class CorefDocumentType(*options*, *metadata*)
    Bases: `pimlico.datatypes.jsondoc.JsonDocumentType`

    process_document(*doc*)

class CorefCorpus(*base_dir*, *pipeline*, ***kwargs*)
    Bases: *pimlico.datatypes.jsondoc.JsonDocumentCorpus*

    datatype_name = 'corenlp_coref'

    data_point_type
        alias of *CorefDocumentType*

class CorefCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
    Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

    document_to_raw_data(*data*)

class Entity(*id*, *mentions*)
    Bases: `object`

class Mention(*id*, *sentence_num*, *start_index*, *end_index*, *text*, *type*, *position=None*, *animacy=None*, *is_representative_mention=None*, *number=None*, *gender=None*)
    Bases: `object`

    static from_json(*json*)

    to_json_dict()

pimlico.datatypes.coref.opennlp module

Datatypes for coreference resolution output. Based on OpenNLP's coref output, so includes all the information provided by that. This is a slight different set of information to CoreNLP. Currently, there's no way to convert between the two datatypes, but in future it will be easy to provide an adapter that carries across the information common to the two (which for most purposes will be sufficient).

class CorefDocumentType(*options*, *metadata*)
    Bases: `pimlico.datatypes.jsondoc.JsonDocumentType`

    process_document(*doc*)

class CorefCorpus(*base_dir*, *pipeline*, ***kwargs*)
    Bases: *pimlico.datatypes.jsondoc.JsonDocumentCorpus*

    datatype_name = 'opennlp_coref'

    data_point_type
        alias of *CorefDocumentType*

class CorefCorpusWriter(*base_dir*, *readable=False*, ***kwargs*)
    Bases: *pimlico.datatypes.jsondoc.JsonDocumentCorpusWriter*

    document_to_raw_data(*data*)

class Entity(*id*, *mentions*, *category=None*, *gender=None*, *gender_prob=None*, *number=None*, *number_prob=None*)
    Bases: `object`

    get_head_word(*pronouns=['i', 'you', 'he', 'she', 'it', 'we', 'they', 'me', 'him', 'her', 'us', 'them', 'myself', 'yourself', 'himself', 'herself', 'itself', 'ourself', 'ourselves', 'themselves', 'my', 'your', 'his', 'its', "it's", 'our', 'their', 'mine', 'yours', 'ours', 'theirs', 'this', 'that', 'those', 'these']*)

Retrieve a head word from the entity's mentions if possible. Returns None if no suitable head word can be found: e.g., if all mentions are pronouns.

Pronouns are filtered out using :data:pimlico.utils.linguistic.ENGLISH_PRONOUNS by default. You can override this with the *pronouns* kwargs. If *pronouns=None*, no filtering is done.

to_json_dict()

static **from_json**(*json*)

static **from_java_object**(*obj*)

class **Mention**(*sentence_num*, *start_index*, *end_index*, *text*, *gender=None*, *gender_prob=None*, *number=None*, *number_prob=None*, *head_start_index=None*, *head_end_index=None*, *name_type=None*)

Bases: `object`

static **from_json**(*json*)

to_json_dict()

static **from_java_object**(*obj*)

Module contents

OpenNLPCorefCorpus
alias of `pimlico.datatypes.coref.opennlp.CorefCorpus`

OpenNLPCorefCorpusWriter
alias of `pimlico.datatypes.coref.opennlp.CorefCorpusWriter`

CoreNLPCorefCorpus
alias of `pimlico.datatypes.coref.corenlp.CorefCorpus`

CoreNLPCorefCorpusWriter
alias of `pimlico.datatypes.coref.corenlp.CorefCorpusWriter`

pimlico.datatypes.formatters package

Submodules

pimlico.datatypes.formatters.features module

class **FeatureListScoreFormatter**(*corpus*)

Bases: `pimlico.cli.browser.formatter.DocumentBrowserFormatter`

DATATYPE
alias of `pimlico.datatypes.features.FeatureListScoreDocumentType`

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

pimlico.datatypes.formatters.tokenized module

class TokenizedDocumentFormatter(*corpus*, *raw_data=False*)
Bases: `pimlico.cli.browser.formatter.DocumentBrowserFormatter`

DATATYPE
alias of `pimlico.datatypes.tokenized.TokenizedDocumentType`

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class CharacterTokenizedDocumentFormatter(*corpus*, *raw_data=False*)
Bases: `pimlico.cli.browser.formatter.DocumentBrowserFormatter`

DATATYPE
alias of `pimlico.datatypes.tokenized.CharacterTokenizedDocumentType`

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class SegmentedLinesFormatter(*corpus*)
Bases: `pimlico.cli.browser.formatter.DocumentBrowserFormatter`

DATATYPE
alias of `pimlico.datatypes.tokenized.SegmentedLinesDocumentType`

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

Module contents

pimlico.datatypes.parse package

Submodules

pimlico.datatypes.parse.candc module

class CandcOutputCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

datatype_name = 'candc_output'

data_point_type
alias of `CandcOutputDocumentType`

class CandcOutputCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

pimlico.datatypes.parse.dependency module

class StanfordDependencyParseCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpus`

datatype_name = 'stanford_dependency_parses'

data_point_type
alias of `StanfordDependencyParseDocumentType`

class StanfordDependencyParseCorpusWriter (*base_dir*, *readable=False*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpusWriter`

document_to_raw_data (*data*)

class CoNLLDependencyParseCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

10-field CoNLL dependency parse format (conllx) – i.e. post parsing.

Fields are: id (int), word form, lemma, coarse POS, POS, features, head (int), dep relation, phead (int), pdeprel

The last two are usually not used.

datatype_name = 'conll_dependency_parses'

data_point_type
alias of `CoNLLDependencyParseDocumentType`

class CoNLLDependencyParseCorpusWriter (*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

document_to_raw_data (*data*)

class CoNLLDependencyParseInputCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

The version of the CoNLL format (conllx) that only has the first 6 columns, i.e. no dependency parse yet annotated.

datatype_name = 'conll_dependency_parse_inputs'

data_point_type
alias of `CoNLLDependencyParseInputDocumentType`

class CoNLLDependencyParseInputCorpusWriter (*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

document_to_raw_data (*data*)

Module contents

TODO Parse tress are temporary implementations that don't actually parse the data, but just split it into
sentences.

class TreeStringsDocumentType (*options*, *metadata*)
Bases: `pimlico.datatypes.documents.RawDocumentType`

process_document (*doc*)

class ConstituencyParseTreeCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Note that this is not fully developed yet. At the moment, you'll just get, for each document, a list of the texts of each tree. In future, they will be better represented.

datatype_name = 'parse_trees'

data_point_type
alias of `TreeStringsDocumentType`

class ConstituencyParseTreeCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

class CandcOutputCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

datatype_name = 'candc_output'

data_point_type
alias of `CandcOutputDocumentType`

class CandcOutputCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

class StanfordDependencyParseCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpus`

datatype_name = 'stanford_dependency_parses'

data_point_type
alias of `StanfordDependencyParseDocumentType`

class StanfordDependencyParseCorpusWriter(*base_dir*, *readable=False*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpusWriter`

document_to_raw_data(*data*)

class CoNLLDependencyParseCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

10-field CoNLL dependency parse format (conllx) – i.e. post parsing.

Fields are: id (int), word form, lemma, coarse POS, POS, features, head (int), dep relation, phead (int), pdeprel

The last two are usually not used.

datatype_name = 'conll_dependency_parses'

data_point_type
alias of `CoNLLDependencyParseDocumentType`

class CoNLLDependencyParseCorpusWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

document_to_raw_data(*data*)

class CoNLLDependencyParseInputCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

The version of the CoNLL format (conllx) that only has the first 6 columns, i.e. no dependency parse yet annotated.

datatype_name = 'conll_dependency_parse_inputs'

data_point_type
alias of CoNLLDependencyParseInputDocumentType

class CoNLLDependencyParseInputCorpusWriter(*base_dir*, ***kwargs*)
Bases: *pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter*

document_to_raw_data(*data*)

Submodules

pimlico.datatypes.arrays module

Wrappers around Numpy arrays and Scipy sparse matrices.

class NumpyArray(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.files.NamedFileCollection*

datatype_name = 'numpy_array'

filenames = ['array.npy']

array

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

class NumpyArrayWriter(*base_dir*, *additional_name=None*)
Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

set_array(*array*)

class ScipySparseMatrix(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Wrapper around Scipy sparse matrices. The matrix loaded is always in COO format – you probably want to convert to something else before using it. See scipy docs on sparse matrix conversions.

datatype_name = 'scipy_sparse_array'

filenames = ['array.mtx']

array

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

class ScipySparseMatrixWriter(*base_dir*, *additional_name=None*)

Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

set_matrix(*mat*)

pimlico.datatypes.base module

Datatypes provide interfaces for reading (and in some cases writing) datasets. At their most basic, they define a way to iterate over a dataset linearly. Some datatypes may also provide other functionality, such as random access or compression.

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. Custom datatypes will be needed for most datasets when they're used as inputs, but as far as possible, these should be converted into standard datatypes like `TarredCorpus`, early in the pipeline.

Note: The following classes were moved to `core` in version 0.6rc

class PimlicoDatatype(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)

Bases: `object`

The abstract superclass of all datatypes. Provides basic functionality for identifying where data should be stored and such.

Datatypes are used to specify the routines for reading the output from modules. They're also used to specify how to read pipeline inputs. Most datatypes that have data simply read it in when required. Some (in particular those used as inputs) need a preparation phase to be run, where the raw data itself isn't sufficient to implement the reading interfaces required. In this case, they should override prepare_data().

Datatypes may require/allow options to be set when they're used to read pipeline inputs. These are specified, in the same way as module options, by input_module_options on the datatype class.

Datatypes may supply a set of additional datatypes. These should be guaranteed to be available if the main datatype is available. They must require no extra processing to be made available, unless that is done on the fly while reading the datatype (like a filter) or while the main datatype is being written.

Additional datatypes can be accessed in config files by specifying the main datatype (as a previous module, optionally with an output name) and the additional datatype name in the form *main_datatype->additional_name*. Multiple additional names may be given, causing the next name to be looked up as an additional datatype of the initially loaded additional datatype. E..g *main_datatype->additional0->additional1*.

To avoid conflicts in the metadata between datatypes using the same directory, datatypes loaded as additional datatypes have their additional name available to them and use it as a prefix to the metadata filename.

If *use_main_metadata=True* on an additional datatype, the same metadata will be read as for the main datatype to which this is an additional datatype.

module is the ModuleInfo instance for the pipeline module that this datatype was produced by. It may be None, if the datatype wasn't instantiated by a module. It is not required to be set if you're instantiating a datatype in some context other than module output. It should generally be set for input datatypes, though, since they are treated as being created by a special input module.

`requires_data_preparation = False`

`input_module_options = {}`
Override to provide shell commands specific to this datatype. Should include the superclass' list.

`shell_commands = []`
List of additional datatypes provided by this one, given as (name, datatype class) pairs. For each of these, a call to *get_additional_datatype(name)* (once the main datatype is ready) should return a datatype instance that is also ready.

`supplied_additional = []`
Most datatype classes define their own type of corpus, which is often a subtype of some other. Some, however, emulate another type and it is that type that should be considered the be the type of the dataset, not the class itself.

For example, TarredCorpusFilter dynamically produces something that looks like a TarredCorpus, and further down the pipeline, if its type is need, it should be considered to be a TarredCorpus.

Most of the time, this can be left empty, but occasionally it needs to be set.

`emulated_datatype = None`

`datatype_name = 'base_datatype'`

`metadata`
Read in metadata from a file in the corpus directory.

Note that this is no longer cached in memory. We need to be sure that the metadata values returned are always up to date with what is on disk, so always re-read the file when we need to get a value from the metadata. Since the file is typically small, this is unlikely to cause a problem. If we decide to return to cacheing the metadata dictionary in future, we will need to make sure that we can never run into problems with out-of-date metadata being returned.

`get_required_paths`()
Returns a list of paths to files that should be available for the data to be read. The base data_ready() implementation checks that these are all available and, if the datatype is used as an input to a pipeline and requires a data preparation routine to be run, data preparation will not be executed until these files are available.

Paths may be absolute or relative. If relative, they refer to files within the data directory and data_ready() will fail if the data dir doesn't exist.

Returns list of absolute or relative paths

`get_software_dependencies`()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

`prepare_data`(*output_dir*, *log*)

classmethod create_from_options(*base_dir*, *pipeline*, *options={}*, *module=None*)

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

get_detailed_status()
Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

classmethod datatype_full_class_name()
The fully qualified name of the class for this datatype, by which it is reference in config files. Generally, datatypes don't need to override this, but type requirements that take the place of datatypes for type checking need to provide it.

instantiate_additional_datatype(*name*, *additional_name*)
Default implementation just assumes the datatype class can be instantiated using the default constructor, with the same base dir and pipeline as the main datatype. Options given to the main datatype are passed down to the additional datatype.

classmethod check_type(*supplied_type*)
Method used by datatype type-checking algorithm to determine whether a supplied datatype (given as a class, which is a subclass of PimlicoDatatype) is compatible with the present datatype, which is being treated as a type requirement.

Typically, the present class is a type requirement on a module input and *supplied_type* is the type provided by a previous module's output.

The default implementation simply checks whether *supplied_type* is a subclass of the present class. Subclasses may wish to impose different or additional checks.

Parameters **supplied_type** – type provided where the present class is required, or datatype instance

Returns True if the check is successful, False otherwise

classmethod type_checking_name()
Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

classmethod full_datatype_name()
Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

class DynamicOutputDatatype
Bases: `object`

Types of module outputs may be specified as a subclass of `PimlicoDatatype`, or alternatively as an *instance* of DynamicOutputType. In this case, get_datatype() is called when the output datatype is needed, passing in the module info instance for the module, so that a specialized datatype can be produced on the basis of options, input types, etc.

The dynamic type must provide certain pieces of information needed for typechecking.

datatype_name = None

get_datatype(*module_info*)

get_base_datatype_class()

If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return the class. By default, it returns None.

If this information is available, it will be used in documentation.

class DynamicInputDatatypeRequirement

Bases: `object`

Types of module inputs may be given as a subclass of *PimlicoDatatype*, a tuple of datatypes, or an instance a DynamicInputDatatypeRequirement subclass. In this case, check_type(supplied_type) is called during typechecking to check whether the type that we've got conforms to the input type requirements.

Additionally, if datatype_doc_info is provided, it is used to represent the input type constraints in documentation.

datatype_doc_info = None

check_type(*supplied_type*)

type_checking_name()

Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Subclasses may want to override this too.

class MultipleInputs(*datatype_requirements*)

Bases: `object`

An input datatype that can be used as an item in a module's inputs, which lets the module accept an unbounded number of inputs, all satisfying the same datatype requirements. When writing the inputs in a config file, they can be specified as a comma-separated list of the usual type of specification (module name, with optional output name). Each item in the list must point to a datatype that satisfies the type-checking.

The list may also include (or entirely consist of) a base module name from the pipeline that has been expanded into multiple modules according to alternative parameters (the type separated by vertical bars, see *Multiple parameter values*). Use the notation **name*, where *name* is the base module name, to denote all of the expanded module names as inputs. These are treated as if you'd written out all of the expanded module names separated by commas.

In a config file, if you need the same input specification to be repeated multiple times in a list, instead of writing it out explicitly you can use a multiplier to repeat it N times by putting `*N` after it. This is particularly useful when `N` is the result of expanding module variables, allowing the number of times an input is repeated to depend on some modvar expression.

When get_input() is called on the module, instead of returning a single datatype, a list of datatypes is returned.

class PimlicoDatatypeWriter(*base_dir*, *additional_name=None*)

Bases: `object`

Abstract base class fo data writer associated with Pimlico datatypes.

require_tasks(**tasks*)

Add a name or multiple names to the list of output tasks that must be completed before writing is finished

task_complete(*task*)

incomplete_tasks

write_metadata()

subordinate_additional_name(*name*)

class IterableCorpus(**args*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatype*

Superclass of all datatypes which represent a dataset that can be iterated over document by document (or datapoint by datapoint - what exactly we're iterating over may vary, though documents are most common).

The actual type of the data depends on the subclass: it could be, e.g. coref output, etc. Information about the type of individual documents is provided by *document_type* and this is used in type checking.

At creation time, length should be provided in the metadata, denoting how many documents are in the dataset.

datatype_name = 'iterable_corpus'

data_point_type
alias of `pimlico.datatypes.documents.RawDocumentType`

shell_commands = [<pimlico.datatypes.base.CountInvalidCmd object>]

get_detailed_status()
Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

classmethod check_type(*supplied_type*)
Override type checking to require that the supplied type have a document type that is compatible with (i.e. a subclass of) the document type of this class.

classmethod type_checking_name()
Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

classmethod full_datatype_name()
Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

process_document_data_with_datatype(*data*)
Applies the corpus' datatype's process_document() method to the raw data :param data: :return:

class IterableCorpusWriter(*base_dir*, *additional_name=None*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

class InvalidDocument(*module_name*, *error_info=None*)
Bases: `object`

Widely used in Pimlico to represent an empty document that is empty not because the original input document was empty, but because a module along the way had an error processing it. Document readers/writers should generally be robust to this and simply pass through the whole thing where possible, so that it's always possible to work out, where one of these pops up, where the error occurred.

static load(*text*)

static invalid_document_or_text(*text*)
If the text represents an invalid document, parse it and return an InvalidDocument object. Otherwise, return the text as is.

exception DatatypeLoadError
Bases: `exceptions.Exception`

exception DatatypeWriteError
Bases: `exceptions.Exception`

load_datatype(*path*)
Try loading a datatype class for a given path. Raises a DatatypeLoadError if it's not a valid datatype path.

pimlico.datatypes.caevo module

class CaevoCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Datatype for Caevo output. The output is stored exactly as it comes out from Caevo, in an XML format. This datatype reads in that XML and provides easy access to its components.

Since we simply store the XML that comes from Caevo, there's no corresponding corpus writer. The data is output using a :class:pimlico.datatypes.tar.TarredCorpusWriter.

data_point_type
alias of `CaevoDocumentType`

pimlico.datatypes.core module

Some basic core datatypes that are commonly used for simple datatypes, file types, etc.

class SingleTextDocument(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: `pimlico.datatypes.files.NamedFileCollection`

datatype_name = 'single_doc'

filenames = ['data.txt']

read_data()

class SingleTextDocumentWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

class Dict(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: `pimlico.datatypes.files.NamedFileCollection`

Simply stores a Python dict, pickled to disk.

datatype_name = 'dict'

filenames = ['data']

data

class DictWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

class StringList(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatype`

Simply stores a Python list of strings, written out to disk in a readable form. Not the most efficient format, but if the list isn't humungous it's OK (e.g. storing vocabularies).

datatype_name = 'string_list'

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

path

data

class StringListWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

pimlico.datatypes.dictionary module

This module implements the concept of Dictionary – a mapping between words and their integer ids.

The implementation is based on Gensim, because Gensim is wonderful and there's no need to reinvent the wheel. We don't use Gensim's data structure directly, because it's unnecessary to depend on the whole of Gensim just for one data structure.

class Dictionary(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatype`

Dictionary encapsulates the mapping between normalized words and their integer ids.

datatype_name = 'dictionary'

get_data()

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

get_detailed_status()
Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

class DictionaryWriter(*base_dir*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

add_documents(*documents*, *prune_at=2000000*)

filter(*threshold=None*, *no_above=None*, *limit=None*)

pimlico.datatypes.documents module

Document types used to represent datatypes of individual documents in an IterableCorpus or subtype.

class DataPointType(*options*, *metadata*)
Bases: `object`

Base data-point type for iterable corpora. All iterable corpora should have data-point types that are subclasses of this.

input_module_options = {}

formatters = []
List of (name, cls_path) pairs specifying a standard set of formatters that the user might want to choose from to view a dataset of this type. The user is not restricted to this set, but can easily choose these by name, instead of specifying a class path themselves. The first in the list is the default used if no formatter is specified. Falls back to DefaultFormatter if empty

class RawDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.documents.DataPointType`

Base document type. All document types for tarred corpora should be subclasses of this.

It may be used itself as well, where documents are just treated as raw data, though most of the time it will be appropriate to use subclasses to provide more information and processing operations specific to the datatype.

The `process_document()` method produces a data structure in the internal format appropriate for the data point type.

# A problem

If a subclassed type produces an internal data structure that does not work as a sub-type (using duck-typing-style inheritance principles) of its parent type, we can run into problems. See [this comment](https://github.com/markgw/pimlico/issues/1#issuecomment-383620759) for discussion of a solution to be introduced.

I therefore am not going to solve this now: you just need to work around it.

process_document(*doc*)

class RawTextDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.documents.TextDocumentType`

Subclass of TextDocumentType used to indicate that the text hasn't been processed (tokenized, etc). Note that text that has been tokenized, parsed, etc does not used subclasses of this type, so they will not be considered compatible if this type is used as a requirement.

pimlico.datatypes.embeddings module

class Vocab(*word*, *index*, *count=0*)

Bases: `object`

A single vocabulary item, used internally for collecting per-word frequency info. A simplified version of Gensim's `Vocab`.

class Embeddings(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.base.PimlicoDatatype`

Datatype to store embedding vectors, together with their words. Based on Gensim's `KeyedVectors` object, but adapted for use in Pimlico and so as not to depend on Gensim. (This means that this can be used more generally for storing embeddings, even when we're not depending on Gensim.)

Provides a method to map to Gensim's `KeyedVectors` type for compatibility.

Doesn't provide all of the functionality of `KeyedVectors`, since the main purpose of this is for storage of vectors and other functionality, like similarity computations, can be provided by utilities or by direct use of Gensim.

vectors

normed_vectors

vector_size

index2vocab

index2word

vocab

word_vec(*word*)

Accept a single word as input. Returns the word's representation in vector space, as a 1D numpy array.

word_vecs(*words*)
Accept multiple words as input. Returns the words' representations in vector space, as a 1D numpy array.

to_keyed_vectors()

class EmbeddingsWriter(*base_dir*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

write_vectors(*arr*)
Write out vectors from a Numpy array

write_word_counts(*word_counts*)
Write out vocab from a list of words with counts.

Parameters word_counts – list of (unicode, int) pairs giving each word and its count. Vocab indices are determined by the order of words

write_vocab_list(*vocab_items*)
Write out vocab from a list of vocab items (see `Vocab`).

Parameters vocab_items – list of ``Vocab``s

write_keyed_vectors(**kvecs*)
Write both vectors and vocabulary straight from Gensim's `KeyedVectors` data structure. Can accept multiple objects, which will then be concatenated in the output.

pimlico.datatypes.features module

class KeyValueListDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

process_document(*doc*)

class KeyValueListCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpus*

datatype_name = 'key_value_lists'

data_point_type
alias of *KeyValueListDocumentType*

class KeyValueListCorpusWriter(*base_dir*, *separator=' '*, *fv_separator='='*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

document_to_raw_data(*data*)

class TermFeatureListDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.features.KeyValueListDocumentType*

process_document(*doc*)

class TermFeatureListCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.features.KeyValueListCorpus*

Special case of KeyValueListCorpus, where one special feature "term" is always present and the other feature types are counts of the occurrence of a particular feature with this term in each data point.

datatype_name = 'term_feature_lists'

data_point_type
alias of *TermFeatureListDocumentType*

class TermFeatureListCorpusWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.features.KeyValueListCorpusWriter`

document_to_raw_data(*data*)

class IndexedTermFeatureListDataPointType(*options*, *metadata*)
Bases: `pimlico.datatypes.documents.DataPointType`

class IndexedTermFeatureListCorpus(**args*, ***kwargs*)
Bases: `pimlico.datatypes.base.IterableCorpus`

Term-feature instances, indexed by a dictionary, so that all that's stored is the indices of the terms and features and the feature counts for each instance. This is iterable, but, unlike TermFeatureListCorpus, doesn't iterate over documents. Now that we've filtered extracted features down to a smaller vocab, we put everything in one big file, with one data point per line.

Since we're now storing indices, we can use a compact format that's fast to read from disk, making iterating over the dataset faster than if we had to read strings, look them up in the vocab, etc.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

data_point_type
alias of `IndexedTermFeatureListDataPointType`

term_dictionary

feature_dictionary

class IndexedTermFeatureListCorpusWriter(*base_dir*, *term_dictionary*, *feature_dictionary*, *bytes=4*, *signed=False*, *index_input=False*, ***kwargs*)
Bases: `pimlico.datatypes.base.IterableCorpusWriter`

index_input=True means that the input terms and feature names are already mapped to dictionary indices, so are assumed to be ints. Otherwise, inputs will be looked up in the appropriate dictionary to get an index.

write_dictionaries()

add_data_points(*iterable*)

class FeatureListScoreDocumentType(*options*, *metadata*)
Bases: `pimlico.datatypes.documents.RawDocumentType`

Document type that stores a list of features, each associated with a floating-point score. The feature lists are simply lists of indices to a feature set for the whole corpus that includes all feature types and which is stored along with the dataset. These may be binary features (present or absent for each data point), or may have a weight associated with them. If they are binary, the returned data will have a weight of 1 associated with each.

A corpus of this type can be used to train, for example, a regression.

If scores and weights are passed in as Decimal objects, they will be stored as strings. If they are floats, they will be converted to Decimals via their string representation (avoiding some of the oddness of converting between binary and decimal representations). To avoid loss of precision, pass in all scores and weights as Decimal objects.

formatters = [('features', 'pimlico.datatypes.formatters.features.FeatureListScoreForm

process_document(*doc*)

class FeatureListScoreCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

datatype_name = 'scored_weight_feature_lists'

data_point_type
alias of *FeatureListScoreDocumentType*

class FeatureListScoreCorpusWriter(*base_dir*, *features*, *separator=':'*, *index_input=False*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

Input should be a list of data points. Each is a (score, feature list) pair, where score is a Decimal, or other numeric type. Feature list is a list of (feature name, weight) pairs, or just feature names. If weights are not given, they will default to 1 when read in (but no weight is stored).

If index_input=True, it is assumed that feature IDs will be given instead of feature names. Otherwise, the feature names will be looked up in the feature list. Any features not found in the feature type list will simply be skipped.

document_to_raw_data(*data*)

pimlico.datatypes.files module

class File(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Simple datatype that supplies a single file, providing the path to it. Use `FileCollection` with a single file where possible.

This is an abstract class: subclasses need to provide a way of getting to (e.g. storing) the filename in question.

This overlaps somewhat with `FileCollection`, but is mainly here for backwards compatibility. Future datatypes should prefer the use of `FileCollection`.

datatype_name = 'file'

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

absolute_path

class NamedFileCollection(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatype*

Abstract base datatype for datatypes that store a fixed collection of files, which have fixed names (or at least names that can be determined from the class). Very many datatypes fall into this category. Overriding this base class provides them with some common functionality, including the possibility of creating a union of multiple datatypes.

The attribute `filenames` should specify a list of filenames contained by the datatype.

All files are contained in the datatypes data directory. If files are stored in subdirectories, this may be specified in the list of filenames using `/` s. (Always use forward slashes, regardless of the operating system.)

datatype_name = 'file_collection'

filenames = []

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

get_absolute_path (*filename*)

absolute_filenames

read_file (*filename=None*, *mode='r'*)

read_files (*mode='r'*)

class NamedFileCollectionWriter (*base_dir*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

filenames = []

write_file (*filename*, *data*)

get_absolute_path (*filename*)

named_file_collection_union (**file_collection_classes*, ***kwargs*)
Takes a number of subclasses of `FileCollection` and produces a new datatype that shares the functionality of all of them and is constituted of the union of the filenames.

The datatype name of the result will be produced automatically from the inputs, unless the kwargs `name` is given to specify a new one.

Note that the input classes' __init__`` s will each be called once, with the standard ``PimlicoDatatype args. If this behaviour does not suit the datatypes you're using, override the init or define the union some other way.

filename_with_range (*val*)
Option processor for file paths with an optional start and end line at the end.

class UnnamedFileCollection (**args*, ***kwargs*)
Bases: `pimlico.datatypes.base.IterableCorpus`

Note: Datatypes used for reading input data are being phased out and replaced by input reader modules. Use `pimlico.modules.input.text.raw_text_files` instead of this for reading raw text files at the start of your pipeline.

A file collection that's just a bunch of files with arbitrary names. The names are not necessarily known until the data is ready. They may be specified as a list in the metadata, or through datatype options, in the case of input datatypes.

This datatype is particularly useful for loading individual files or sets of files at the start of a pipeline. If you just want the raw data from each file, you can use this class as it is. It's an `IterableCorpus` with a raw data type. If you want to apply some special processing to each file, do so by overriding this class and specifying the `data_point_type`, providing a `DataPointType` subclass that does the necessary processing.

When using it as an input datatype to load arbitrary files, specify a list of absolute paths to the files you want to use. They 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.

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.

The same postprocessing will be applied to every file. In cases where you need to apply different processing to different subsets of the files, define multiple input modules, with different data point types, for example, and then combine them using `pimlico.modules.corpora.concat`.

`datatype_name = 'unnamed_file_collection'`

`input_module_options = {'exclude': {'type': <function _fn at 0x7f703d7516e0>, 'help'`

`data_ready`()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

`get_paths`(*error_on_missing=False*)

`get_paths_from_options`(*error_on_missing=False*)
Get a list of paths to all the files specified in the `files` option. If `error_on_missing=True`, non-optional paths or globs that do not correspond to an existing file cause an IOError to be raised.

`path_name_to_doc_name`(*path*)

`class UnnamedFileCollectionWriter`(**args*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

Use as a context manager to write a bag of files out to the output directory.

Provide each file's raw data and a filename to use to the function *write_file()* within the *with* statement. The writer will keep track of what files you've output and store the list.

`get_absolute_path`(*filename*)

`add_written_file`(*filename*)
Add a filename to the list of files included in the collection. Should only be called after the file of that name has been written to the path given by *get_absolute_path()*.

Usually, you should use *write_file()* instead, which handles this itself.

`write_file`(*filename*, *data*)
Write data to a file and add the file to the collection.

`NamedFile`(*name*)
Datatype factory that produces something like a `File` datatype, pointing to a single file, but doesn't store its path, just refers to a particular file in the data dir.

Parameters **`name`** – name of the file

Returns datatype class

`class FilesInput`(*min_files=1*)
Bases: `pimlico.datatypes.base.DynamicInputDatatypeRequirement`

`datatype_doc_info = 'A file collection containing at least one file (or a given specif`

`check_type`(*supplied_type*)

`FileInput`
alias of `pimlico.datatypes.files.FilesInput`

`class NamedFileWriter`(*base_dir*, *filename*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

`absolute_path`

`write_data`(*data*)
Write the given string data to the appropriate output file

class RawTextFiles(**args*, ***kwargs*)
Bases: *pimlico.datatypes.files.UnnamedFileCollection*

Essentially the same as RawTextDirectory, but more flexible. Should generally be used in preference to RawTextDirectory.

Basic datatype for reading in all the files in a collection as raw text documents.

Generally, this may be appropriate to use as the input datatype at the start of a pipeline. You'll then want to pass it through a tarred corpus filter to get it into a suitable form for input to other modules.

data_point_type
alias of *pimlico.datatypes.documents.RawTextDocumentType*

class RawTextDirectory(**args*, ***kwargs*)
Bases: *pimlico.datatypes.base.IterableCorpus*

Basic datatype for reading in all the files in a directory and its subdirectories as raw text documents.

Generally, this may be appropriate to use as the input datatype at the start of a pipeline. You'll then want to pass it through a tarred corpus filter to get it into a suitable form for input to other modules.

datatype_name = 'raw_text_directory'

input_module_options = {'encoding': {'default': 'utf8', 'help': "Encoding used to s

data_point_type
alias of *pimlico.datatypes.documents.RawTextDocumentType*

requires_data_preparation = True

prepare_data(*output_dir*, *log*)

walk()

filter_document(*doc*)
Each document is passed through this filter before being yielded. Default implementation does nothing, but this makes it easy to implement custom postprocessing by overriding.

get_required_paths()
Returns a list of paths to files that should be available for the data to be read. The base data_ready() implementation checks that these are all available and, if the datatype is used as an input to a pipeline and requires a data preparation routine to be run, data preparation will not be executed until these files are available.

Paths may be absolute or relative. If relative, they refer to files within the data directory and data_ready() will fail if the data dir doesn't exist.

Returns list of absolute or relative paths

pimlico.datatypes.floats module

Similar to :mod:pimlico.datatypes.ints, but for lists of floats.

class FloatListsDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

formatters = [('float_lists', 'pimlico.datatypes.floats.FloatListsFormatter')]

process_document(*data*)

read_rows(*reader*)

class FloatListsFormatter(*corpus*)
Bases: *pimlico.cli.browser.formatter.DocumentBrowserFormatter*

DATATYPE
alias of *FloatListsDocumentType*

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

class FloatListsDocumentCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpus*

Corpus of float list data: each doc contains lists of float. Unlike *IntegerTableDocumentCorpus*, they are not all constrained to have the same length. The downside is that the storage format (and probably I/O speed) isn't quite as efficient. It's still better than just storing ints as strings or JSON objects.

The floats are stored as C double, which use 8 bytes. At the moment, we don't provide any way to change this. An alternative would be to use C floats, losing precision but (almost) halving storage size.

datatype_name = 'float_lists_corpus'

data_point_type
alias of *FloatListsDocumentType*

class FloatListsDocumentCorpusWriter(*base_dir*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

document_to_raw_data(*data*)

class FloatListDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

Like FloatListsDocumentType, but each document is treated as a single list of floats.

process_document(*data*)

read_floats(*reader*)

class FloatListDocumentCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpus*

Corpus of float data: each doc contains a single sequence of floats.

The floats are stored as C doubles, using 8 bytes each.

datatype_name = 'float_list_corpus'

data_point_type
alias of *FloatListDocumentType*

class FloatListDocumentCorpusWriter(*base_dir*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

document_to_raw_data(*data*)

pimlico.datatypes.ints module

class IntegerListsDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

unpacker

process_document (*data*)

read_rows (*reader*)

class IntegerListsDocumentCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Corpus of integer list data: each doc contains lists of ints. Unlike `IntegerTableDocumentCorpus`, they are not all constrained to have the same length. The downside is that the storage format (and probably I/O speed) isn't quite as good. It's still better than just storing ints as strings or JSON objects.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

datatype_name = 'integer_lists_corpus'

data_point_type
alias of `IntegerListsDocumentType`

class IntegerListsDocumentCorpusWriter (*base_dir*, *signed=False*, *bytes=8*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data (*data*)

class IntegerListDocumentType (*options*, *metadata*)
Bases: `pimlico.datatypes.documents.RawDocumentType`

Like IntegerListsDocumentType, but each document is treated as a single list of integers.

unpacker

int_size

process_document (*data*)

read_ints (*reader*)

class IntegerListDocumentCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Corpus of integer data: each doc contains a single sequence of ints.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

datatype_name = 'integer_list_corpus'

data_point_type
alias of `IntegerListDocumentType`

class IntegerListDocumentCorpusWriter (*base_dir*, *signed=False*, *bytes=8*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data (*data*)

pimlico.datatypes.jsondoc module

class JsonDocumentCorpus (*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Very simple document corpus in which each document is a JSON object.

datatype_name = 'json'

data_point_type
alias of `JsonDocumentType`

class JsonDocumentCorpusWriter(*base_dir*, *readable=False*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

If readable=True, JSON text output will be nicely formatted so that it's human-readable. Otherwise, it will be formatted to take up less space.

document_to_raw_data(*data*)

pimlico.datatypes.keras module

Datatypes for storing and loading Keras models.

class KerasModelWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

Writer for storing both types of Keras model (since they provide the same storage interface).

write_model(*model*)

write_architecture(*model*)

write_weights(*model*)

class KerasModel(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatype`

Datatype for both types of Keras models, stored using Keras' own storage mechanisms. This uses Keras' method of storing the model architecture as JSON and stores the weights using hdf5.

datatype_name = 'keras_model'

custom_objects = {}

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

get_custom_objects()

load_model()

class KerasModelBuilderClassWriter(*base_dir*, *build_params*, *builder_class_path*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

Writer for storing Keras models in the manner described in :cls:KerasModelBuilderClass.

write_weights(*model*)

class KerasModelBuilderClass(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.base.PimlicoDatatype`

An alternative way to store Keras models.

Create a class whose init method build the model architecture. It should take a kwarg called *build_params*, which is a JSON-encodable dictionary of parameters that determine how the model gets build (hyperparameters). When you initialize your model for training, create this hyperparameter dictionary and use it to instantiate the model class.

Use the KerasModelBuilderClassWriter to store the model during training. Create a writer, then start model training, storing the weights to the filename given by the *weights_filename* attribute of the writer. The hyperparameter dictionary will also be stored.

The writer also stores the fully-qualified path of the model-builder class. When we read the datatype and want to rebuild the model, we import the class, instantiate it and then set its weights to those we've stored.

The model builder class must have the model stored in an attribute *model*.

weights_filename

load_build_params()

create_builder_class(*override_params=None*)

load_model(*override_params=None*)

Instantiate the model builder class with the stored parameters and set the weights on the model to those stored.

Returns model builder instance (keras model in attribute *model*

pimlico.datatypes.plotting module

class PlotOutput(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)

Bases: `pimlico.datatypes.base.PimlicoDatatype`

Output from matplotlib plotting.

Contains the dataset being plotted, a script to build the plot, and the output PDF.

Also supplies additional datatypes to point to the individual files.

supplied_additional = [('pdf', <class 'pimlico.datatypes.files.NamedFile'>), ('code',

script_path

plot()

Runs the plotting script. Errors are not caught, so if there's a problem in the script they'll be raised.

pdf_path

data_path

class PlotOutputWriter(*base_dir*)

Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

pimlico.datatypes.r module

class RTabSeparatedValuesFile(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)

Bases: `pimlico.datatypes.files.File`

Tabular data stored in a TSV file, suitable for reading in using R's *read.delim* function.

datatype_name = 'r_tsv'

absolute_path

get_detailed_status()

Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

class RTabSeparatedValuesFileWriter(*base_dir*, *headings=None*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

Writer for TSV files suitable for reading with R.

If *headings* is specified, this is written as the first line of the file, so *headings=TRUE* should be used when reading into R.

Double quotes (") in the fields will be replaced by double-double quotes (""), which R interprets as a double quote. Fields containing tabs will be surrounded by normal double quotes. When you read the data into R, the default value of *quotes* (") should therefore be fine. No escaping is performed on single quotes (').

absolute_path

write_row(*row*)

If elements are not of string type, they will be coerced to a string for writing. If you want to format them differently, do it before calling this method and pass in strings.

pimlico.datatypes.results module

class NumericResult(*base_dir*, *pipeline*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatype*

Simple datatype to contain a numeric value and a label, representing the result of some process, such as evaluation of a model on a task.

For example, allows results to be plotted by passing them into a graph plotting module.

datatype_name = 'numeric_result'

data

Raw JSON data

result

The numeric result being stored

label

A label to identify this result (e.g. model name)

class NumericResultWriter(*base_dir*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

pimlico.datatypes.sklearn module

Datatypes for storing and loading Scikit-learn models.

class SklearnModelWriter(*base_dir*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

write_model (*model*)

class SklearnModel (*base_dir*, *pipeline*, ***kwargs*)

Bases: *pimlico.datatypes.base.PimlicoDatatype*

Datatype for storing Scikit-learn models.

Very simple storage mechanism: we just pickle the model to a file. Instead of the standard Python pickle package, we use Joblib, which stores large data objects (especially Numpy arrays) more efficiently.

get_software_dependencies ()

Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

load_model ()

pimlico.datatypes.spans module

class SentenceSpansDocumentType (*options*, *metadata*)

Bases: pimlico.datatypes.jsondoc.JsonDocumentType

process_document (*doc*)

class SentenceSpansCorpus (*base_dir*, *pipeline*, ***kwargs*)

Bases: *pimlico.datatypes.jsondoc.JsonDocumentCorpus*

data_point_type

alias of *SentenceSpansDocumentType*

class SentenceSpansCorpusWriter (*base_dir*, *readable=False*, ***kwargs*)

Bases: *pimlico.datatypes.jsondoc.JsonDocumentCorpusWriter*

document_to_raw_data (*data*)

pimlico.datatypes.table module

get_struct (*bytes*, *signed*, *row_length*)

class IntegerTableDocumentType (*options*, *metadata*)

Bases: *pimlico.datatypes.documents.RawDocumentType*

unpacker

row_size

process_document (*data*)

read_rows (*reader*)

class IntegerTableDocumentCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

Corpus of tabular integer data: each doc contains rows of ints, where each row contains the same number of values. This allows a more compact representation, which doesn't require converting the ints to strings or scanning for line ends, so is quite a bit quicker and results in much smaller file sizes. The downside is that the files are not human-readable.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

datatype_name = 'integer_table_corpus'

data_point_type
alias of `IntegerTableDocumentType`

class IntegerTableDocumentCorpusWriter(*base_dir*, *row_length*, *signed=False*, *bytes=8*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

pimlico.datatypes.tar module

class TarredCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.base.IterableCorpus`

datatype_name = 'tar'

document_preprocessors = []

data_point_type
alias of `pimlico.datatypes.documents.RawDocumentType`

extract_file(*archive_name*, *filename*)
Extract an individual file by archive name and filename. This is not an efficient way of extracting a lot of files. The typical use case of a tarred corpus is to iterate over its files, which is much faster.

doc_iter(*subsample=None*, *start_after=None*, *skip=None*)

archive_iter(*subsample=None*, *start_after=None*, *skip=None*)

process_document(*data*)
Process the data read in for a single document. Allows easy implementation of datatypes using TarredCorpus to do all the archive handling, etc, just specifying a particular way of handling the data within documents.

By default, uses the document data processing provided by the document type.

Most of the time, you shouldn't need to override this, but just write a document type that does the necessary processing.

I think we should remove this once the new (forthcoming) datatype system is ready, but we'll need to check that there's not still a use case for it.

list_archive_iter()

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

class TarredCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpusWriter`

If gzip=True, each document is gzipped before adding it to the archive. Not the same as creating a tarball, since the docs are gzipped *before* adding them, not the whole archive together, but it means we can easily iterate over the documents, unzipping them as required.

A subtlety of TarredCorpusWriter and its subclasses is that, as soon as the writer has been initialized, it must be legitimate to initialize a datatype to read the corpus. Naturally, at this point there will be no documents in the corpus, but it allows us to do document processing on the fly by initializing writers and readers to be sure the pre/post-processing is identical to if we were writing the docs to disk and reading them in again.

If append=True, existing archives and their files are not overwritten, the new files are just added to the end. This is useful where we want to restart processing that was broken off in the middle. If trust_length=True, when appending the initial length of the corpus is read from the metadata already written. Otherwise (default), the number of docs already written is actually counted during initialization. This is sensible when the previous writing process may have ended abruptly, so that the metadata is not reliable. If you know you can trust the metadata, however, setting trust_length=True will speed things up.

add_document(*archive_name*, *doc_name*, *data*)

document_to_raw_data(*doc*)

Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class AlignedTarredCorpora(*corpora*)

Bases: `object`

Iterator for iterating over multiple corpora simultaneously that contain the same files, grouped into archives in the same way. This is the standard utility for taking multiple inputs to a Pimlico module that contain different data but for the same corpus (e.g. output of different tools).

archive_iter(*subsample=None*, *start_after=None*)

exception CorpusAlignmentError

Bases: `exceptions.Exception`

exception TarredCorpusIterationError

Bases: `exceptions.Exception`

pimlico.datatypes.tokenized module

class TokenizedDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.documents.TextDocumentType`

formatters = [('tokenized_doc', 'pimlico.datatypes.formatters.tokenized.TokenizedDocume

process_document(*doc*, *as_type=None*)

class TokenizedCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpus`

Specialized datatype for a tarred corpus that's had tokenization applied. The datatype does very little - the main reason for its existence is to allow modules to require that a corpus has been tokenized before it's given as input.

Each document is a list of sentences. Each sentence is a list of words.

datatype_name = 'tokenized'

data_point_type
alias of *TokenizedDocumentType*

class TokenizedCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

Simple writer that takes lists of tokens and outputs them with a sentence per line and tokens separated by spaces.

document_to_raw_data(*doc*)
Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class CharacterTokenizedDocumentType(*options*, *metadata*)
Bases: `pimlico.datatypes.tokenized.TokenizedDocumentType`

Simple character-level tokenized corpus. The text isn't stored in any special way, but is represented when read internally just as a sequence of characters in each sentence.

If you need a more sophisticated way to handle character-type (or any non-word) units within each sequence, see *SegmentedLinesDocumentType*.

formatters = [('char_tokenized_doc', 'pimlico.datatypes.formatters.tokenized.Character

process_document(*doc*, *as_type=None*)

class CharacterTokenizedCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

Simple writer that takes lists of char-tokens and outputs them with a sentence per line. Just joins together all the characters to store the sentence, since they can be divided up again when read.

document_to_raw_data(*doc*)
Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class SegmentedLinesDocumentType(*options*, *metadata*)
Bases: `pimlico.datatypes.tokenized.TokenizedDocumentType`

Document consisting of lines, each split into elements, which may be characters, words, or whatever. Rather like a tokenized corpus, but doesn't make the assumption that the elements (words in the case of a tokenized corpus) don't include spaces.

You might use this, for example, if you want to train character-level models on a text corpus, but don't use strictly single-character units, perhaps grouping together certain short character sequences.

Uses the character `/` to separate elements. If a `/` is found in an element, it is stored as *@slash@*, so this string is assumed not to be used in any element (which seems reasonable enough, generally).

formatters = [('segmented_lines', 'pimlico.datatypes.formatters.tokenized.SegmentedLin

process_document(*doc*, *as_type=None*)

class SegmentedLinesCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*doc*)
Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the

actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

pimlico.datatypes.vrt module

class VRTWord(*word*, **attributes*)
Bases: `object`

Word with all its annotations.

The Korp docs give the following example list of positional attributes (columns):

> word form, the number of the token within the sentence, lemma, lemma with compound boundaries marked, part of speech, morphological analysis, dependency head number and dependency relation

However, they are not fixed and different files may have different numbers of attributes with different meanings. This information is not included in the data file.

class VRTText(*words*, *paragraph_ranges=[]*, *sentence_ranges=[]*, *opening_tag=None*)
Bases: `object`

Contains a single VRT text (i.e. document).

Note that VRT's structures are not hierarchical: they can be overlapping. See VRT docs.

We don't currently process structural attributes. This can easily be added later if necessary.

static from_string(*data*)

paragraphs

sentences

word_strings

class VRTDocumentType(*options*, *metadata*)
Bases: *`pimlico.datatypes.documents.DataPointType`*

Document type for annotation text documents read in from VRT files (VeRticalized Text, as used by Korp:).

formatters = [('vrt', 'pimlico.datatypes.vrt.VRTFormatter')]

process_document(*doc*)

class VRTFormatter(*corpus*)
Bases: *`pimlico.cli.browser.formatter.DocumentBrowserFormatter`*

DATATYPE
alias of *`VRTDocumentType`*

format_document(*doc*)
Format a single document and return the result as a string (or unicode, but it will be converted to ASCII for display).

Must be overridden by subclasses.

pimlico.datatypes.word2vec module

class Word2VecModel(*base_dir*, *pipeline*, ***kwargs*)
Bases: *`pimlico.datatypes.base.PimlicoDatatype`*

Datatype for storing Gensim-trained word2vec embeddings.

See also:

Datatype ***pimlico.datatypes.embeddings.Embeddings*** Another, more generic way, to write the same data, which should generally be used in preference to this one. `Embeddings` does not depend on Gensim, but can be converted to Gensim's data structure easily.

shell_commands = [<pimlico.datatypes.word2vec.NearestNeighboursCommand object>, <pimli

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

load_model()

model

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

class Word2VecModelWriter(*base_dir*, *verb_only=False*, ***kwargs*)
Bases: *pimlico.datatypes.base.PimlicoDatatypeWriter*

Note: Generally, it's preferable to use *pimlico.datatypes.embeddings.Embeddings*, which is more generic, so easier to connect up with general vector/embedding-handling modules.

write_word2vec_model(*model*)

write_keyed_vectors(*vectors*)

pimlico.datatypes.word_annotations module

class WordAnnotationsDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

sentence_boundary_re

word_boundary

word_re

process_document(*raw_data*)

class WordAnnotationCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpus*

datatype_name = 'word_annotations'

data_point_type
alias of `WordAnnotationsDocumentType`

annotation_fields = None

read_annotation_fields()
Get the available annotation fields from the dataset's configuration. These are the actual fields that will be available in the dictionary produced corresponding to each word.

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

class WordAnnotationCorpusWriter(*sentence_boundary*, *word_boundary*, *word_format*, *non-word_chars*, *base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

Ensures that the correct metadata is provided for a word annotation corpus. Doesn't take care of the formatting of the data: that needs to be done by the writing code, or by a subclass.

class SimpleWordAnnotationCorpusWriter(*base_dir*, *field_names*, *field_sep=u'|'*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

Takes care of writing word annotations in a simple format, where each line contains a sentence, words are separated by spaces and a series of annotation fields for each word are separated by |s (or a given separator). This corresponds to the standard tag format for C&C.

document_to_raw_data(*data*)

class AddAnnotationField(*input_name*, *add_fields*)
Bases: `pimlico.datatypes.base.DynamicOutputDatatype`

get_datatype(*module_info*)

classmethod get_base_datatype_class()
If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return the class. By default, it returns None.

If this information is available, it will be used in documentation.

class WordAnnotationCorpusWithRequiredFields(*required_fields*)
Bases: `pimlico.datatypes.base.DynamicInputDatatypeRequirement`

Dynamic (functional) type that can be used in place of a module's input type. In typechecking, checks whether the input module is a WordAnnotationCorpus (or subtype) and whether its fields include all of those required.

check_type(*supplied_type*)

exception AnnotationParseError
Bases: `exceptions.Exception`

pimlico.datatypes.xml module

Input datatype for extracting documents from XML files. Gigaword, for example, is stored in this way.

Depends on BeautifulSoup (see "bs4" target in lib dir Makefile).

DEPRECATED: Use input module `pimlico.modules.input.xml` instead. Input datatypes are being phased out.

class XmlDocumentIterator(**args*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpus`

requires_data_preparation = True

input_module_options = {'document_name_attr': {'default': 'id', 'help': "Attribute

data_point_type

alias of `pimlico.datatypes.documents.RawTextDocumentType`

get_software_dependencies()

Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

prepare_data(*output_dir*, *log*)

data_ready()

Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

Module contents

OpenNLPCorefCorpus

alias of `pimlico.datatypes.coref.opennlp.CorefCorpus`

OpenNLPCorefCorpusWriter

alias of `pimlico.datatypes.coref.opennlp.CorefCorpusWriter`

CoreNLPCorefCorpus

alias of `pimlico.datatypes.coref.corenlp.CorefCorpus`

CoreNLPCorefCorpusWriter

alias of `pimlico.datatypes.coref.corenlp.CorefCorpusWriter`

class ConstituencyParseTreeCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpus`

Note that this is not fully developed yet. At the moment, you'll just get, for each document, a list of the texts of each tree. In future, they will be better represented.

datatype_name = 'parse_trees'

data_point_type

alias of `TreeStringsDocumentType`

class ConstituencyParseTreeCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

class TreeStringsDocumentType(*options*, *metadata*)
Bases: `pimlico.datatypes.documents.RawDocumentType`

process_document(*doc*)

class CandcOutputCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpus`

datatype_name = 'candc_output'

data_point_type
alias of `CandcOutputDocumentType`

class CandcOutputCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

class StanfordDependencyParseCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpus`

datatype_name = 'stanford_dependency_parses'

data_point_type
alias of `StanfordDependencyParseDocumentType`

class StanfordDependencyParseCorpusWriter(*base_dir*, *readable=False*, ***kwargs*)
Bases: `pimlico.datatypes.jsondoc.JsonDocumentCorpusWriter`

document_to_raw_data(*data*)

class CoNLLDependencyParseCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

10-field CoNLL dependency parse format (conllx) – i.e. post parsing.

Fields are: id (int), word form, lemma, coarse POS, POS, features, head (int), dep relation, phead (int), pdeprel

The last two are usually not used.

datatype_name = 'conll_dependency_parses'

data_point_type
alias of `CoNLLDependencyParseDocumentType`

class CoNLLDependencyParseCorpusWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

document_to_raw_data(*data*)

class CoNLLDependencyParseInputCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpus`

The version of the CoNLL format (conllx) that only has the first 6 columns, i.e. no dependency parse yet annotated.

datatype_name = 'conll_dependency_parse_inputs'

data_point_type
alias of `CoNLLDependencyParseInputDocumentType`

class CoNLLDependencyParseInputCorpusWriter(*base_dir*, ***kwargs*)
Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

document_to_raw_data(*data*)

class NumpyArray(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.files.NamedFileCollection`

datatype_name = 'numpy_array'

filenames = ['array.npy']

array

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

class NumpyArrayWriter(*base_dir*, *additional_name=None*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

set_array(*array*)

class ScipySparseMatrix(*base_dir*, *pipeline*, ***kwargs*)
Bases: `pimlico.datatypes.base.PimlicoDatatype`

Wrapper around Scipy sparse matrices. The matrix loaded is always in COO format – you probably want to convert to something else before using it. See scipy docs on sparse matrix conversions.

datatype_name = 'scipy_sparse_array'

filenames = ['array.mtx']

array

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

class ScipySparseMatrixWriter(*base_dir*, *additional_name=None*)
Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

set_matrix(*mat*)

class PimlicoDatatype(*base_dir*, *pipeline*, *module=None*, *additional_name=None*, *use_main_metadata=False*, ***kwargs*)
Bases: `object`

The abstract superclass of all datatypes. Provides basic functionality for identifying where data should be stored and such.

Datatypes are used to specify the routines for reading the output from modules. They're also used to specify how to read pipeline inputs. Most datatypes that have data simply read it in when required. Some (in particular those used as inputs) need a preparation phase to be run, where the raw data itself isn't sufficient to implement the reading interfaces required. In this case, they should override prepare_data().

Datatypes may require/allow options to be set when they're used to read pipeline inputs. These are specified, in the same way as module options, by input_module_options on the datatype class.

Datatypes may supply a set of additional datatypes. These should be guaranteed to be available if the main datatype is available. They must require no extra processing to be made available, unless that is done on the fly while reading the datatype (like a filter) or while the main datatype is being written.

Additional datatypes can be accessed in config files by specifying the main datatype (as a previous module, optionally with an output name) and the additional datatype name in the form *main_datatype->additional_name*. Multiple additional names may be given, causing the next name to be looked up as an additional datatype of the initially loaded additional datatype. E..g *main_datatype->additional0->additional1*.

To avoid conflicts in the metadata between datatypes using the same directory, datatypes loaded as additional datatypes have their additional name available to them and use it as a prefix to the metadata filename.

If *use_main_metadata=True* on an additional datatype, the same metadata will be read as for the main datatype to which this is an additional datatype.

module is the ModuleInfo instance for the pipeline module that this datatype was produced by. It may be None, if the datatype wasn't instantiated by a module. It is not required to be set if you're instantiating a datatype in some context other than module output. It should generally be set for input datatypes, though, since they are treated as being created by a special input module.

`requires_data_preparation = False`

`input_module_options = {}`

Override to provide shell commands specific to this datatype. Should include the superclass' list.

`shell_commands = []`

List of additional datatypes provided by this one, given as (name, datatype class) pairs. For each of these, a call to *get_additional_datatype(name)* (once the main datatype is ready) should return a datatype instance that is also ready.

`supplied_additional = []`

Most datatype classes define their own type of corpus, which is often a subtype of some other. Some, however, emulate another type and it is that type that should be considered the be the type of the dataset, not the class itself.

For example, TarredCorpusFilter dynamically produces something that looks like a TarredCorpus, and further down the pipeline, if its type is need, it should be considered to be a TarredCorpus.

Most of the time, this can be left empty, but occasionally it needs to be set.

`emulated_datatype = None`

`datatype_name = 'base_datatype'`

`metadata`

Read in metadata from a file in the corpus directory.

Note that this is no longer cached in memory. We need to be sure that the metadata values returned are always up to date with what is on disk, so always re-read the file when we need to get a value from the metadata. Since the file is typically small, this is unlikely to cause a problem. If we decide to return to cacheing the metadata dictionary in future, we will need to make sure that we can never run into problems with out-of-date metadata being returned.

get_required_paths()
Returns a list of paths to files that should be available for the data to be read. The base data_ready() implementation checks that these are all available and, if the datatype is used as an input to a pipeline and requires a data preparation routine to be run, data preparation will not be executed until these files are available.

Paths may be absolute or relative. If relative, they refer to files within the data directory and data_ready() will fail if the data dir doesn't exist.

Returns list of absolute or relative paths

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

prepare_data(*output_dir*, *log*)

classmethod create_from_options(*base_dir*, *pipeline*, *options={}*, *module=None*)

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

get_detailed_status()
Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

classmethod datatype_full_class_name()
The fully qualified name of the class for this datatype, by which it is reference in config files. Generally, datatypes don't need to override this, but type requirements that take the place of datatypes for type checking need to provide it.

instantiate_additional_datatype(*name*, *additional_name*)
Default implementation just assumes the datatype class can be instantiated using the default constructor, with the same base dir and pipeline as the main datatype. Options given to the main datatype are passed down to the additional datatype.

classmethod check_type(*supplied_type*)
Method used by datatype type-checking algorithm to determine whether a supplied datatype (given as a class, which is a subclass of PimlicoDatatype) is compatible with the present datatype, which is being treated as a type requirement.

Typically, the present class is a type requirement on a module input and *supplied_type* is the type provided by a previous module's output.

The default implementation simply checks whether *supplied_type* is a subclass of the present class. Subclasses may wish to impose different or additional checks.

Parameters **`supplied_type`** – type provided where the present class is required, or datatype instance

Returns True if the check is successful, False otherwise

classmethod type_checking_name()

Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

classmethod full_datatype_name()

Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

class PimlicoDatatypeWriter(*base_dir*, *additional_name=None*)

Bases: `object`

Abstract base class fo data writer associated with Pimlico datatypes.

require_tasks(**tasks*)

Add a name or multiple names to the list of output tasks that must be completed before writing is finished

task_complete(*task*)

incomplete_tasks

write_metadata()

subordinate_additional_name(*name*)

class IterableCorpus(**args*, ***kwargs*)

Bases: `pimlico.datatypes.base.PimlicoDatatype`

Superclass of all datatypes which represent a dataset that can be iterated over document by document (or data-point by datapoint - what exactly we're iterating over may vary, though documents are most common).

The actual type of the data depends on the subclass: it could be, e.g. coref output, etc. Information about the type of individual documents is provided by *document_type* and this is used in type checking.

At creation time, length should be provided in the metadata, denoting how many documents are in the dataset.

datatype_name = 'iterable_corpus'

data_point_type

alias of `pimlico.datatypes.documents.RawDocumentType`

shell_commands = [<pimlico.datatypes.base.CountInvalidCmd object>]

get_detailed_status()

Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

classmethod check_type(*supplied_type*)

Override type checking to require that the supplied type have a document type that is compatible with (i.e. a subclass of) the document type of this class.

classmethod type_checking_name()

Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Classes that override check_supplied_type() may want to override this too.

classmethod full_datatype_name()

Returns a string/unicode name for the datatype that includes relevant sub-type information. The default implementation just uses the attribute *datatype_name*, but subclasses may have more detailed information to add. For example, iterable corpus types also supply information about the data-point type.

process_document_data_with_datatype(*data*)

Applies the corpus' datatype's process_document() method to the raw data :param data: :return:

class IterableCorpusWriter(*base_dir*, *additional_name=None*)

Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

class DynamicOutputDatatype

Bases: `object`

Types of module outputs may be specified as a subclass of `PimlicoDatatype`, or alternatively as an *instance* of DynamicOutputType. In this case, get_datatype() is called when the output datatype is needed, passing in the module info instance for the module, so that a specialized datatype can be produced on the basis of options, input types, etc.

The dynamic type must provide certain pieces of information needed for typechecking.

datatype_name = None

get_datatype(*module_info*)

get_base_datatype_class()

If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return the class. By default, it returns None.

If this information is available, it will be used in documentation.

class DynamicInputDatatypeRequirement

Bases: `object`

Types of module inputs may be given as a subclass of `PimlicoDatatype`, a tuple of datatypes, or an instance a DynamicInputDatatypeRequirement subclass. In this case, check_type(supplied_type) is called during typechecking to check whether the type that we've got conforms to the input type requirements.

Additionally, if datatype_doc_info is provided, it is used to represent the input type constraints in documentation.

datatype_doc_info = None

check_type(*supplied_type*)

type_checking_name()

Supplies a name for this datatype to be used in type-checking error messages. Default implementation just provides the class name. Subclasses may want to override this too.

class InvalidDocument(*module_name*, *error_info=None*)

Bases: `object`

Widely used in Pimlico to represent an empty document that is empty not because the original input document was empty, but because a module along the way had an error processing it. Document readers/writers should generally be robust to this and simply pass through the whole thing where possible, so that it's always possible to work out, where one of these pops up, where the error occurred.

static load(*text*)

static invalid_document_or_text(*text*)

If the text represents an invalid document, parse it and return an InvalidDocument object. Otherwise, return the text as is.

exception DatatypeLoadError

Bases: `exceptions.Exception`

exception DatatypeWriteError

Bases: `exceptions.Exception`

load_datatype(*path*)

Try loading a datatype class for a given path. Raises a DatatypeLoadError if it's not a valid datatype path.

class MultipleInputs(*datatype_requirements*)

Bases: `object`

An input datatype that can be used as an item in a module's inputs, which lets the module accept an unbounded number of inputs, all satisfying the same datatype requirements. When writing the inputs in a config file, they can be specified as a comma-separated list of the usual type of specification (module name, with optional output name). Each item in the list must point to a datatype that satisfies the type-checking.

The list may also include (or entirely consist of) a base module name from the pipeline that has been expanded into multiple modules according to alternative parameters (the type separated by vertical bars, see *Multiple parameter values*). Use the notation **name*, where *name* is the base module name, to denote all of the expanded module names as inputs. These are treated as if you'd written out all of the expanded module names separated by commas.

In a config file, if you need the same input specification to be repeated multiple times in a list, instead of writing it out explicitly you can use a multiplier to repeat it N times by putting `*N` after it. This is particularly useful when `N` is the result of expanding module variables, allowing the number of times an input is repeated to depend on some modvar expression.

When get_input() is called on the module, instead of returning a single datatype, a list of datatypes is returned.

class CaevoCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: *`pimlico.datatypes.tar.TarredCorpus`*

Datatype for Caevo output. The output is stored exactly as it comes out from Caevo, in an XML format. This datatype reads in that XML and provides easy access to its components.

Since we simply store the XML that comes from Caevo, there's no corresponding corpus writer. The data is output using a :class:pimlico.datatypes.tar.TarredCorpusWriter.

data_point_type

alias of `CaevoDocumentType`

class Dictionary(*base_dir*, *pipeline*, ***kwargs*)

Bases: *`pimlico.datatypes.base.PimlicoDatatype`*

Dictionary encapsulates the mapping between normalized words and their integer ids.

datatype_name = 'dictionary'

get_data()

data_ready()

Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

get_detailed_status()

Returns a list of strings, containing detailed information about the data. Only called if data_ready() == True.

Subclasses may override this to supply useful (human-readable) information specific to the datatype. They should called the super method.

class DictionaryWriter(*base_dir*)

Bases: `pimlico.datatypes.base.PimlicoDatatypeWriter`

add_documents(*documents*, *prune_at=2000000*)

filter(*threshold=None*, *no_above=None*, *limit=None*)

class KeyValueListCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpus`

datatype_name = 'key_value_lists'

data_point_type

alias of `KeyValueListDocumentType`

class KeyValueListCorpusWriter(*base_dir*, *separator=' '*, *fv_separator='='*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

document_to_raw_data(*data*)

class KeyValueListDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.documents.RawDocumentType`

process_document(*doc*)

class TermFeatureListCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.features.KeyValueListCorpus`

Special case of KeyValueListCorpus, where one special feature "term" is always present and the other feature types are counts of the occurrence of a particular feature with this term in each data point.

datatype_name = 'term_feature_lists'

data_point_type

alias of `TermFeatureListDocumentType`

class TermFeatureListCorpusWriter(*base_dir*, ***kwargs*)

Bases: `pimlico.datatypes.features.KeyValueListCorpusWriter`

document_to_raw_data(*data*)

class TermFeatureListDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.features.KeyValueListDocumentType`

process_document(*doc*)

class IndexedTermFeatureListCorpus(**args*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpus`

Term-feature instances, indexed by a dictionary, so that all that's stored is the indices of the terms and features and the feature counts for each instance. This is iterable, but, unlike TermFeatureListCorpus, doesn't iterate over documents. Now that we've filtered extracted features down to a smaller vocab, we put everything in one big file, with one data point per line.

Since we're now storing indices, we can use a compact format that's fast to read from disk, making iterating over the dataset faster than if we had to read strings, look them up in the vocab, etc.

By default, the ints are stored as C longs, which use 4 bytes. If you know you don't need ints this big, you can choose 1 or 2 bytes, or even 8 (long long). By default, the ints are unsigned, but they may be signed.

data_point_type
alias of *IndexedTermFeatureListDataPointType*

term_dictionary

feature_dictionary

class IndexedTermFeatureListCorpusWriter(*base_dir*, *term_dictionary*, *feature_dictionary*, *bytes=4*, *signed=False*, *index_input=False*, ***kwargs*)
Bases: *pimlico.datatypes.base.IterableCorpusWriter*

index_input=True means that the input terms and feature names are already mapped to dictionary indices, so are assumed to be ints. Otherwise, inputs will be looked up in the appropriate dictionary to get an index.

write_dictionaries()

add_data_points(*iterable*)

class IndexedTermFeatureListDataPointType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.DataPointType*

class FeatureListScoreCorpus(*base_dir*, *pipeline*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpus*

datatype_name = 'scored_weight_feature_lists'

data_point_type
alias of *FeatureListScoreDocumentType*

class FeatureListScoreCorpusWriter(*base_dir*, *features*, *separator=':'*, *index_input=False*, ***kwargs*)
Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

Input should be a list of data points. Each is a (score, feature list) pair, where score is a Decimal, or other numeric type. Feature list is a list of (feature name, weight) pairs, or just feature names. If weights are not given, they will default to 1 when read in (but no weight is stored).

If index_input=True, it is assumed that feature IDs will be given instead of feature names. Otherwise, the feature names will be looked up in the feature list. Any features not found in the feature type list will simply be skipped.

document_to_raw_data(*data*)

class FeatureListScoreDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.RawDocumentType*

Document type that stores a list of features, each associated with a floating-point score. The feature lists are simply lists of indices to a feature set for the whole corpus that includes all feature types and which is stored along with the dataset. These may be binary features (present or absent for each data point), or may have a weight associated with them. If they are binary, the returned data will have a weight of 1 associated with each.

A corpus of this type can be used to train, for example, a regression.

If scores and weights are passed in as Decimal objects, they will be stored as strings. If they are floats, they will be converted to Decimals via their string representation (avoiding some of the oddness of converting between binary and decimal representations). To avoid loss of precision, pass in all scores and weights as Decimal objects.

formatters = [('features', 'pimlico.datatypes.formatters.features.FeatureListScoreForm

process_document(*doc*)

class JsonDocumentCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpus`

Very simple document corpus in which each document is a JSON object.

datatype_name = 'json'

data_point_type

alias of `JsonDocumentType`

class JsonDocumentCorpusWriter(*base_dir*, *readable=False*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

If readable=True, JSON text output will be nicely formatted so that it's human-readable. Otherwise, it will be formatted to take up less space.

document_to_raw_data(*data*)

class TarredCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpus`

datatype_name = 'tar'

document_preprocessors = []

data_point_type

alias of `pimlico.datatypes.documents.RawDocumentType`

extract_file(*archive_name*, *filename*)

Extract an individual file by archive name and filename. This is not an efficient way of extracting a lot of files. The typical use case of a tarred corpus is to iterate over its files, which is much faster.

doc_iter(*subsample=None*, *start_after=None*, *skip=None*)

archive_iter(*subsample=None*, *start_after=None*, *skip=None*)

process_document(*data*)

Process the data read in for a single document. Allows easy implementation of datatypes using TarredCorpus to do all the archive handling, etc, just specifying a particular way of handling the data within documents.

By default, uses the document data processing provided by the document type.

Most of the time, you shouldn't need to override this, but just write a document type that does the necessary processing.

I think we should remove this once the new (forthcoming) datatype system is ready, but we'll need to check that there's not still a use case for it.

list_archive_iter()

data_ready()

Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

class TarredCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpusWriter`

If gzip=True, each document is gzipped before adding it to the archive. Not the same as creating a tarball, since the docs are gzipped *before* adding them, not the whole archive together, but it means we can easily iterate over the documents, unzipping them as required.

A subtlety of TarredCorpusWriter and its subclasses is that, as soon as the writer has been initialized, it must be legitimate to initialize a datatype to read the corpus. Naturally, at this point there will be no documents in the corpus, but it allows us to do document processing on the fly by initializing writers and readers to be sure the pre/post-processing is identical to if we were writing the docs to disk and reading them in again.

If append=True, existing archives and their files are not overwritten, the new files are just added to the end. This is useful where we want to restart processing that was broken off in the middle. If trust_length=True, when appending the initial length of the corpus is read from the metadata already written. Otherwise (default), the number of docs already written is actually counted during initialization. This is sensible when the previous writing process may have ended abruptly, so that the metadata is not reliable. If you know you can trust the metadata, however, setting trust_length=True will speed things up.

`add_document`(*archive_name*, *doc_name*, *data*)

`document_to_raw_data`(*doc*)
Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

`class AlignedTarredCorpora`(*corpora*)
Bases: `object`

Iterator for iterating over multiple corpora simultaneously that contain the same files, grouped into archives in the same way. This is the standard utility for taking multiple inputs to a Pimlico module that contain different data but for the same corpus (e.g. output of different tools).

`archive_iter`(*subsample=None*, *start_after=None*)

`exception CorpusAlignmentError`
Bases: `exceptions.Exception`

`exception TarredCorpusIterationError`
Bases: `exceptions.Exception`

`class TokenizedDocumentType`(*options*, *metadata*)
Bases: `pimlico.datatypes.documents.TextDocumentType`

`formatters = [('tokenized_doc', 'pimlico.datatypes.formatters.tokenized.TokenizedDocume`

`process_document`(*doc*, *as_type=None*)

`class TokenizedCorpus`(*base_dir*, *pipeline*, ***kwargs*)
Bases: *`pimlico.datatypes.tar.TarredCorpus`*

Specialized datatype for a tarred corpus that's had tokenization applied. The datatype does very little - the main reason for its existence is to allow modules to require that a corpus has been tokenized before it's given as input.

Each document is a list of sentences. Each sentence is a list of words.

`datatype_name = 'tokenized'`

`data_point_type`
alias of *`TokenizedDocumentType`*

`class TokenizedCorpusWriter`(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)
Bases: *`pimlico.datatypes.tar.TarredCorpusWriter`*

Simple writer that takes lists of tokens and outputs them with a sentence per line and tokens separated by spaces.

`document_to_raw_data`(*doc*)
Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the

actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class SegmentedLinesDocumentType(*options*, *metadata*)

Bases: *pimlico.datatypes.tokenized.TokenizedDocumentType*

Document consisting of lines, each split into elements, which may be characters, words, or whatever. Rather like a tokenized corpus, but doesn't make the assumption that the elements (words in the case of a tokenized corpus) don't include spaces.

You might use this, for example, if you want to train character-level models on a text corpus, but don't use strictly single-character units, perhaps grouping together certain short character sequences.

Uses the character / to separate elements. If a / is found in an element, it is stored as *@slash@*, so this string is assumed not to be used in any element (which seems reasonable enough, generally).

formatters = [('segmented_lines', 'pimlico.datatypes.formatters.tokenized.SegmentedLin

process_document(*doc*, *as_type=None*)

class SegmentedLinesCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)

Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

document_to_raw_data(*doc*)

Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class CharacterTokenizedDocumentType(*options*, *metadata*)

Bases: *pimlico.datatypes.tokenized.TokenizedDocumentType*

Simple character-level tokenized corpus. The text isn't stored in any special way, but is represented when read internally just as a sequence of characters in each sentence.

If you need a more sophisticated way to handle character-type (or any non-word) units within each sequence, see *SegmentedLinesDocumentType*.

formatters = [('char_tokenized_doc', 'pimlico.datatypes.formatters.tokenized.Character

process_document(*doc*, *as_type=None*)

class CharacterTokenizedCorpusWriter(*base_dir*, *gzip=False*, *append=False*, *trust_length=False*, *encoding='utf-8'*, ***kwargs*)

Bases: *pimlico.datatypes.tar.TarredCorpusWriter*

Simple writer that takes lists of char-tokens and outputs them with a sentence per line. Just joins together all the characters to store the sentence, since they can be divided up again when read.

document_to_raw_data(*doc*)

Overridden by subclasses to provide the mapping from the structured data supplied to the writer to the actual raw string to be written to disk. Override this instead of add_document(), so that filters can do the mapping on the fly without writing the output to disk.

class WordAnnotationCorpus(*base_dir*, *pipeline*, ***kwargs*)

Bases: *pimlico.datatypes.tar.TarredCorpus*

datatype_name = 'word_annotations'

data_point_type

alias of *WordAnnotationsDocumentType*

annotation_fields = None

read_annotation_fields()

Get the available annotation fields from the dataset's configuration. These are the actual fields that will be available in the dictionary produced corresponding to each word.

data_ready()

Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

class WordAnnotationCorpusWriter(*sentence_boundary*, *word_boundary*, *word_format*, *nonword_chars*, *base_dir*, ***kwargs*)

Bases: `pimlico.datatypes.tar.TarredCorpusWriter`

Ensures that the correct metadata is provided for a word annotation corpus. Doesn't take care of the formatting of the data: that needs to be done by the writing code, or by a subclass.

class SimpleWordAnnotationCorpusWriter(*base_dir*, *field_names*, *field_sep=u'|'*, ***kwargs*)

Bases: `pimlico.datatypes.word_annotations.WordAnnotationCorpusWriter`

Takes care of writing word annotations in a simple format, where each line contains a sentence, words are separated by spaces and a series of annotation fields for each word are separated by |s (or a given separator). This corresponds to the standard tag format for C&C.

document_to_raw_data(*data*)

class AddAnnotationField(*input_name*, *add_fields*)

Bases: `pimlico.datatypes.base.DynamicOutputDatatype`

get_datatype(*module_info*)

classmethod get_base_datatype_class()

If it's possible to say before the instance of a ModuleInfo is available what base datatype will be produced, implement this to return the class. By default, it returns None.

If this information is available, it will be used in documentation.

class WordAnnotationCorpusWithRequiredFields(*required_fields*)

Bases: `pimlico.datatypes.base.DynamicInputDatatypeRequirement`

Dynamic (functional) type that can be used in place of a module's input type. In typechecking, checks whether the input module is a WordAnnotationCorpus (or subtype) and whether its fields include all of those required.

check_type(*supplied_type*)

exception AnnotationParseError

Bases: `exceptions.Exception`

class WordAnnotationsDocumentType(*options*, *metadata*)

Bases: `pimlico.datatypes.documents.RawDocumentType`

sentence_boundary_re

word_boundary

word_re

process_document(*raw_data*)

class XmlDocumentIterator(**args*, ***kwargs*)

Bases: `pimlico.datatypes.base.IterableCorpus`

requires_data_preparation = True

input_module_options = {'document_name_attr': {'default': 'id', 'help': "Attribute

data_point_type
alias of *pimlico.datatypes.documents.RawTextDocumentType*

get_software_dependencies()
Check that all software required to read this datatype is installed and locatable. This is separate to metadata config checks, so that you don't need to satisfy the dependencies for all modules in order to be able to run one of them. You might, for example, want to run different modules on different machines. This is called when a module is about to be executed and each of the dependencies is checked.

Returns a list of instances of subclasses of :class:~pimlico.core.dependencies.base.SoftwareDependency, representing the libraries that this module depends on.

Take care when providing dependency classes that you don't put any import statements at the top of the Python module that will make loading the dependency type itself dependent on runtime dependencies. You'll want to run import checks by putting import statements within this method.

You should call the super method for checking superclass dependencies.

prepare_data(*output_dir*, *log*)

data_ready()
Check whether the data corresponding to this datatype instance exists and is ready to be read.

Default implementation just checks whether the data dir exists. Subclasses might want to add their own checks, or even override this, if the data dir isn't needed.

class DataPointType(*options*, *metadata*)
Bases: `object`

Base data-point type for iterable corpora. All iterable corpora should have data-point types that are subclasses of this.

input_module_options = {}

formatters = []
List of (name, cls_path) pairs specifying a standard set of formatters that the user might want to choose from to view a dataset of this type. The user is not restricted to this set, but can easily choose these by name, instead of specifying a class path themselves. The first in the list is the default used if no formatter is specified. Falls back to DefaultFormatter if empty

class RawDocumentType(*options*, *metadata*)
Bases: *pimlico.datatypes.documents.DataPointType*

Base document type. All document types for tarred corpora should be subclasses of this.

It may be used itself as well, where documents are just treated as raw data, though most of the time it will be appropriate to use subclasses to provide more information and processing operations specific to the datatype.

The `process_document()` method produces a data structure in the internal format appropriate for the data point type.

# A problem

If a subclassed type produces an internal data structure that does not work as a sub-type (using duck-typing-style inheritance principles) of its parent type, we can run into problems. See [this comment](https://github.com/markgw/pimlico/issues/1#issuecomment-383620759) for discussion of a solution to be introduced.

I therefore am not going to solve this now: you just need to work around it.

process_document(*doc*)

class RawTextDocumentType(*options*, *metadata*)
Bases: `pimlico.datatypes.documents.TextDocumentType`

Subclass of TextDocumentType used to indicate that the text hasn't been processed (tokenized, etc). Note that text that has been tokenized, parsed, etc does not used subclasses of this type, so they will not be considered compatible if this type is used as a requirement.

pimlico.test package

Submodules

pimlico.test.pipeline module

Pipeline tests

Pimlico modules and datatypes cannot always be easily tested with unit tests and where they can it's often not easy to work out how to write the tests in a neatly packaged way. Instead, modules can package up tests in the form of a small pipeline that comes with a tiny dataset to use as input. The pipeline can be run in a test environment, where software dependencies are installed and local config is prepared to store output and so on.

This way of providing tests also has the advantage that modules at the same time provide a demo (or several) of how to use them – how pipeline config should look and what sort of input data to use.

class TestPipeline(*pipeline*, *run_modules*, *log*)
Bases: `object`

static load_pipeline(*path*, *storage_root*)
Load a test pipeline from a config file.

Path may be absolute, or given relative to Pimlico test data directory (`PIMLICO_ROOT/test/data`)

get_uninstalled_dependencies()

test_all_modules()

test_input_module(*module_name*)

test_module_execution(*module_name*)

run_test_pipeline(*path*, *module_names*, *log*, *no_clean=False*)
Run a test pipeline, loading the pipeline config from a given path (which may be relative to the Pimlico test data directory) and running each of the named modules, including any of those modules' dependencies.

Any software dependencies not already available that can be installed automatically will be installed in the current environment. If there are unsatisfied dependencies that can't be automatically installed, an error will be raised.

If any of the modules name explicitly is an input dataset, it is loaded and data_ready() is checked. If it is an IterableCorpus, it is tested simply by iterating over the full corpus.

run_test_suite(*pipelines_and_modules*, *log*, *no_clean=False*)

Parameters pipeline_and_modules – list of (pipeline, modules) pairs, where pipeline is a path to a config file and modules a list of module names to test

clear_storage_dir()

exception TestPipelineRunError
Bases: `exceptions.Exception`

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*)

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: `object`

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: `object`

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property.

Often useful in Pimlico datatypes, where it can be time-consuming to load data, but we can't do it once when the datatype is first loaded, since the data might not be ready at that point. Instead, we can access the data, or particular parts of it, using properties and easily cache the result.

Taken from: https://github.com/bottlepy/bottle

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: `object`

classmethod from_local_config(*local_config*)

send_pimlico_email(*subject*, *content*, *local_config*, *log*)
Primary method for sending emails from Pimlico. Tries to send an email with the given content, using the email details found in the local config. If something goes wrong, an error is logged on the given log.

Parameters

- **subject** – email subject
- **content** – email text (may be unicode)
- **local_config** – local config dictionary
- **log** – logger to log errors to (and info if the sending works)

send_text_email(*email_config*, *subject*, *content=None*)

exception EmailError
Bases: `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*)
Utility for copying a large directory and displaying a progress bar showing how much is copied.

Parameters

- **source_dir** –
- **target_dir** –

Returns

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. 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: `object`

Direct a readable output (e.g. pipe from a subprocess) to a queue. Returns the queue. Output is added to the queue one line at a time. To perform a non-blocking read call get_nowait() or get(timeout=T)

`get_nowait`()

`get`(*timeout=None*)

`get_available`()
Don't block. Just return everything that's available in the queue.

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.

class SafeProgressBar(*maxval=None*, *widgets=None*, *term_width=None*, *poll=1*, *left_justify=True*, *fd=<open file '<stderr>', mode 'w'>*)

Bases: `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: `object`

Passed in to ProgressBar instead of a file descriptor (e.g. stderr) to ensure that nothing gets output.

read(*size=None*)

readLine(*size=None*)

write(*s*)

close()

class NonOutputtingProgressBar(**args*, ***kwargs*)

Bases: `pimlico.utils.progress.SafeProgressBar`

Behaves like ProgressBar, but doesn't output anything.

class LittleOutputtingProgressBar(**args*, ***kwargs*)

Bases: `pimlico.utils.progress.SafeProgressBar`

Behaves like ProgressBar, but doesn't output much. Instead of constantly redrawing the progress bar line, it outputs a simple progress message every time it hits the next 10% mark.

If running on a terminal, this will update the line, as with a normal progress bar. If piping to a file, this will just print a new line occasionally, so won't fill up your file with thousands of progress updates.

start()

Starts measuring time, and prints the bar at 0%.

It returns self so you can use it like this: >>> pbar = ProgressBar().start() >>> for i in range(100): … # do something … pbar.update(i+1) … >>> pbar.finish()

finish()

Puts the ProgressBar bar in the finished state.

slice_progress(*iterable*, *num_items*, *title=None*)

class ProgressBarIter(*iterable*, *title=None*)

Bases: `object`

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.web module

download_file(*url*, *target_file*)

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 Future plans

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

1.6.1 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: 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/v0.8/docs/core/dependencies.rst, line 73.)

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

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/core/dependencies.rst, line 80.)

Todo: Write documentation for this

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/core/module_structure.
line 9.)

Todo: Document variants

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/core/variants.rst,
line 5.)

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/v0.8/docs/guides/map_module.rst
line 5.)

Todo: Use a dataset that everyone can get to in the example

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/guides/setup.rst,
line 84.)

Todo: Add more output convertors: currently only provides tokenization

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 26.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Replace check_runtime_dependencies() with get_software_dependencies()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 17.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Replace check_runtime_dependencies() with get_software_dependencies()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 17.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

Todo: Document this module

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pimlico/checkouts/v0.8/docs/modules/pimlico.modu
line 12.)

1.6.2 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.

1.6.3 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.

1.6.4 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

Index