Plumbium Documentation

Release 0.10.0

Jon Stutters

January 09, 2017

Contents

Plumbium is a Python package for wrapping scripts so that their inputs and outputs are preserved in a consistent way and results are recorded.

CHAPTER 1

Why?

Does your directory listing look like this?

```
jstutters@dirac ~/my_study % ll *
-rw-rw-r--. 1 jstutters staff 0 Apr 15 10:31 bad_results.txt
-rw-rw-r--. 1 jstutters staff 0 Apr 15 10:32 old_method.xls
-rw-rw-r--. 1 jstutters staff 0 Apr 15 10:31 results.txt
-rw-rw-r--. 1 jstutters staff 0 Apr 15 10:31 use_this.data.csv

01:
total 8.0K
drwxrwxr-x. 2 jstutters staff 2 Apr 15 10:34 001
drwxrwxr-x. 2 jstutters staff 2 Apr 15 10:34 001-dont_use
drwxrwxr-x. 2 jstutters staff 2 Apr 15 10:34 002
drwxrwxr-x. 2 jstutters staff 2 Apr 15 10:34 002-good
drwxrwxr-x. 2 jstutters staff 2 Apr 15 10:34 003
-rw-rw-r--. 1 jstutters staff 0 Apr 15 10:36 subject1_t1_dont_change.nii.gz
```

When an analysis is run with Plumbium all the input files are copied to a temporary directory in which the analysis is run. When the analysis has finished all the files created are collected into an archive and saved along with all the printed outputs from the analysis stages and any exceptions that occurred. Plumbium can also record results to a database or spreadsheet. To find out more, read the tutorial or dive into the API documentation.

1.1 Example

```
from plumbium import call, record, pipeline
from plumbium.artefacts import TextFile


@record()
def pipeline_stage_1(f):
    call(['/bin/cat', f.filename])


@record()
def pipeline_stage_2(f):
    call(['/bin/cat', f.filename])


def my_pipeline(file1, file2):
    pipeline_stage_1(file1)
```

```
    pipeline_stage_2(file2)


def example_pipeline():
    pipeline.run(
        'example',
        my_pipeline,
        '/my/data/directory',
        TextFile('month00/data.txt'), TextFile('month12/data.txt')
    )


if __name__ == '__main__':
    example_pipeline()
```

CHAPTER 2

Learn more

2.1 Installation

2.1.1 Requirements

Plumbium is tested with Python v2.7 - 3.5. Use of the MongoDB or SQLDatabase result recorders requires the installation of the `pymongo` or `sqlalchemy` modules as appropriate. Depending on your database SQLAlchemy may require additional support libraries to be installed.

2.1.2 Latest stable release

Plumbium is hosted on PyPI. The recommended installation method is to run `pip install plumbium`

2.1.3 Development version

The development version of Plumbium is available on Github.

```
git clone https://github.com/jstutters/plumbium.git
cd plumbium
pip install .
```

2.2 Tutorial

2.2.1 Imports

To use Plumbium you should start by importing the *call* function, *record* decorator, *pipeline* instance and any *plumbium.artefacts* you need. Artefacts are classes representing the data files used by your pipeline e.g. text files and images.

```
from plumbium import call, record, pipeline
from plumbium.artefacts import TextFile
```

2.2.2 Processing stages

Next, define the stages of your analysis to be recorded. For this example we'll concatenate two files in the first stage and then count the words of the resulting file in the second stage. The *record* decorator indicates that the function should be recorded. The list of arguments to record is used to name the return values from the function - the number of arguments to record should match the number of variables returned the the function. Calls to external programs should be made using *call* so that printed output can be captured.

```
@record('concatenated_file')
def concatenate(input_1, input_2):
    cmd = 'cat {0.filename} {1.filename} > joined.txt'.format(
        input_1, input_2
    )
    call([cmd], shell=True)
    return TextFile('joined.txt')

@record(count)
def count_words(target):
    wc_output = call(['wc', target.filename])
    return int(wc_output.strip())
```

2.2.3 The complete pipeline

Now to use our stages to define the whole pipeline. Functions decorated with *record* return an instance of *ProcessOutput*, the outputs from the function can be accessed using a dict-like method.

```
def cat_and_count(input_1, input_2):
    concatenate_output = concatenate(input_1, input_2)
    count_output = count_words(concatenate_output['concatenated_file'])
    return count_output['count']
```

2.2.4 Running the pipeline

Finally we use *pipeline.run* to execute the pipeline.

```
import sys

if __name__ == '__main__':
    input_1 = TextFile(sys.argv[1])
    input_2 = TextFile(sys.argv[2])
    pipeline.run('cat_and_count', cat_and_count, '.', input_1, input_2)
```

To try this out save the complete example as tutorial.py, create a pair of text files in the same directory and then run `python tutorial.py [text file 1] [text file 2]`. If everything works no errors should be printed and a file called `cat_and_count-[date]_[time].tar.gz` should be created.

2.2.5 Results

Extract the result file using `tar -zxf [result file]` and have a look in the new directory. You'll find the two files that you used as input to the script, the result output of concatenating the files as `joined.txt` and a `.json` file. If you open the `.json` file you'll see a full record of the commands run (any errors that occur will also be recorded in this file).

```
{
    "processes": [
        {
            "function": "concatenate",
            "returned": [
                "TextFile('joined.txt')"
            ],
            "input_kwargs": {},
            "finish_time": "20160426 12:13",
            "start_time": "20160426 12:13",
            "printed_output": "",
            "input_args": [
                "TextFile('text_file.txt')",
                "TextFile('text_file2.txt')"
            ],
            "called_commands": [
                "cat text_file.txt text_file2.txt"
            ]
        },
        {
            "function": "count_words",
            "returned": [],
            "input_kwargs": {},
            "finish_time": "20160426 12:13",
            "start_time": "20160426 12:13",
            "printed_output": "4 joined.txt\n",
            "input_args": [
                "TextFile('joined.txt')"
            ],
            "called_commands": [
                "wc joined.txt"
            ]
        }
    ],
    "name": "cat_and_count",
    "finish_date": "20160426 12:13",
    "start_date": "20160426 12:13",
    "results": {
        "0": 1234
    },
    "dir": ".",
    "inputs: [
        "TextFile('text_file.txt')",
        "TextFile('text_file2.txt')"
    ],
    "environment": {
        "python_packages": [
            ...
        ],
        "hostname": "machine.example.com",
        "environ": {
            ...
        },
        "uname": [
            "Linux"
            "machine.example.com"
            "3.10.0-327.18.2.el7.x86_64",
            "#1 SMP Thu May 12 11:03:55 UTC 2016",
```

```
            "x86_64"
        ]
    }
}
```

2.3 Next steps

2.3.1 Adding metadata

Often it is useful to add extra information to an analysis record such as software versions or patient identification numbers. This information can be added to an analysis using the `metadata` keyword argument.

```
pipeline.run(
    'example',
    my_pipeline,
    base_directory,
    metadata={'site': 5, 'subject': 1, 'version': '0.1beta2'}
)
```

This metadata dictionary will be included in the saved JSON file and can be used by result recorders and to name output files.

2.3.2 Pipeline result names

If your pipeline function returns values these can be named in the report file using the `result_names` keyword argument.

```
pipeline.run(
    'example',
    my_pipeline,
    base_directory,
    result_names=('foo', 'bar')
)
```

2.3.3 Output file naming

By default the results of an analysis run are saved as `'[analysis_name]-[start date]_[start_time].tar.gz'`. This behaviour can be changed by adding the `filename` keyword to your *`pipeline.run`* call.

```
pipeline.run(
    'example',
    my_pipeline,
    base_directory,
    metadata={'site': 5, 'subject': 1},
    filename='{name}-{metadata[site]:03d}-{metadata[subject]:02d}-{start_date:%Y%m%d}'
)
```

The filename argument should be given as a string using Python's format string syntax. When the file is saved the fields in this string will be replaced using the results structure - the layout of this structure can be seen by inspecting the JSON file that Plumbium produces.

2.3.4 Recording results

In addition to archiving analysis results to a file Plumbium can record analysis outcomes to a number of other destinations.

CSV file

The *CSVFile* recorder outputs selected fields from the results structure to a CSV file (which will be created or appended to as appropriate). To use CSVFile first create an instance of the class.

```
csvfile = CSVFile(
    'csv_results.csv',
    OrderedDict([
        ('start_date', lambda x: x['start_date']),
        ('data_val', lambda x: x['processes'][-1]['printed_output'].strip().split(' ')[0])
    ])
)
```

The first argument is the path of the CSV file you want to record to. The second argument is a dictionary consisting of keys corresponding to the column names in your CSV file and function which will return the appropriate value for each column. An `OrderedDict` should be used so that the columns are ordered as expected (using a regular *dict* will give a random order of columns.

SQL database

To record to any SQL database supported by SQLAlchemy use the *SQLDatabase* class.

```
db = SQLDatabase(
    'sqlite:///db.sqlite',
    'results',
    {
        'wordcount': lambda x: x['processes'][-1]['printed_output'].strip().split(' ')[0],
        'start_date': lambda x: x['start_date']
    }
)
```

The first argument should be a database URL in a form recognised by SQLAlchemy, the second argument is the name of the database table to insert the new result into (this table must exist - Plumbium won't try to create it), the last argument is a dictionary of column names and functions to output values as described above.

MongoDB

Plumbium can save the complete JSON result structure to a MongoDB server using the *MongoDB* class.

```
mongodb = MongoDB('mongodb://localhost:27017/', 'plumbium', 'results')
```

The first arugment is a MongoDB URL (see the PyMongo tutorial for details). The second argument is the database name and the final argument is the collection to insert into.

Slack

The Slack recorder allows a message to be sent to a Slack channel configured with a Webhook. You will need the name of the channel to post to and the Webhook URL from the Slack website.

```
slack = Slack(
    'https://hooks.slack.com/services/...',
    '#channel',
    OrderedDict([
        ('start_date', lambda x: x['start_date']),
        ('data_val', lambda x: x['processes'][-1]['printed_output'].strip().split(' ')[0])
    ])
)
```

The first argument is the Webhook URL, the second is the channel to post to (the channel name should include the preceding #). The example shown will send a message like the following to Slack upon completion:

> Plumbium task complete
> start date: 20160101 11:59
> data_val: 55

2.4 Contribute

- Issue Tracker: github.com/jstutters/plumbium/issues
- Source Code: github.com/jstutters/plumbium

2.5 Support

If you are having problems, please let me know by submitting an issue in the tracker.

2.6 Modules

plumbium is a module for recording the activity of file processing pipelines. Main plumbium module containing the Pipeline class and function recording methods.

class plumbium.processresult.**OutputRecorder**
Holds commands used via the call function and their resulting output.

reset()
Clear the stored commands and output.

class plumbium.processresult.**Pipeline**
Main class managing the recording of a processing pipeline.

record(*process*)
Record a process in this pipeline.

Parameters **process** (*plumbium.processresult.ProcessOutput*) – The new result.

run(*name*, *pipeline_func*, *base_dir*, **inputs*, ***kwargs*)
Execute a function as a recorded pipeline.

Parameters

- **name** (*str*) – The name of the pipeline - used to name the output file.
- **pipeline_function** (*function*) – The function to be run.

- **base_dir** (*str*) – The directory in which to save the pipeline output, also used as the root directory for input filenames if the filenames given are not absolute.
- ***inputs** – The inputs to the pipeline.

Keyword Arguments

- **metadata** (*dict*) – Additional information to be included in the result JSON.
- **filename** (*str*) – String template for the result filename.
- **result_recorder** (*object*) – An instance of a class implementing a *write()* method that accepts the report dictionary.
- **result_names** (*str*) – An iterable of strings containing the names for any values returned by the pipeline.
- **report_name** (*str*) – Filename for the JSON report (default: *report.json*).

save(*exception=None*, *report_name='report.json'*)
Save a record of the pipeline execution.

Creates a JSON file with information about the pipeline then saves it to a gzipped tar file along with all files used in the pipeline.

Keyword Arguments exception (`exceptions.Exception` or *None*) – The exception which caused the pipeline run to fail

class `plumbium.processresult.`**ProcessOutput**(*func*, *args*, *kwargs*, *commands*, *output*, *exception*, *started*, *finished*, ***output_images*)
A record of one stage within a pipeline.

Parameters

- **func** (*function*) – The function that was run.
- **args** (*list*) – The arguments passed to the function.
- **kwargs** (*dict*) – The keyword arguments passed to the function.
- **output** (*str*) – Text printed to stdout or stderr during execution.
- **exception** (`exceptions.Exception` or *None*) – The exception that occurred running the stage if applicable.
- **started** (`datetime.datetime`) – When the stage was started.
- **finished** (`datetime.datetime`) – When the stage finished executing.
- ****output_images** (*plumbium.artefacts.Artefact*) – Images produced by the stage.

__getitem__(*key*)
Get the item corresponding to `key` in the `_results` dictionary.

__iter__()
Get an iterable over the keys in the `_results` dictionary.

__len__()
Get the length of the `_results` dictionary.

as_dict()
Serialize this output as a `dict`.

`plumbium.processresult.`**call**(*cmd*, *cwd=None*, *shell=False*)
Execute scripts and applications in a pipeline with output capturing.

Parameters

- **cmd** (*list*) – List containing the program to be called and any arguments e.g. `['tar', '-x', '-f', 'file.tgz']`.
- **cwd** (*str*) – Working directory in which to execute the command.
- **shell** (*bool*) – Execute the command in a shell.

Returns The output from the called command on stdout and stderr.

Return type str

plumbium.processresult.**record**(**output_names*)

Decorator for wrapping pipeline stages.

Parameters ***output_names** (*str*) – The names of each returned variable.

Module containing the *plumbium.artefacts.Artefact* base class and subclasses.

class plumbium.artefacts.**Artefact**(*filename*, *extension*, *exists=True*)

Base class for Plumbium artefacts (files consumed by and generated by processes).

Parameters

- **filename** (*str*) – The filename of the artefact.
- **extension** (*str*) – The extension of the artefact's filename.

Keyword Arguments **exists** (*boolean*) – If true raise an exception if the file does not exist.

Raises

- `exceptions.ValueError` – If `filename` does not end with `extension`.
- `exceptions.IOError` – If `filename` does not exist.

abspath

The file's absolute path.

basename

The filename without the extension and directory components.

```
>> Artefact('/dir/file.txt').basename
'/dir/file'
```

checksum()

Calculate the SHA-1 checksum of the file.

dereference()

Remove any directory components from the filename.

```
>> a = Artefact('/dir/file.txt')
>> a.dereference()
>> a.filename
'file.txt'
```

dirname

Return the directory component of the filename.

```
>> Artefact('/dir/file.txt').dirname()
'/dir'
```

exists()

Return `True` if *Artefact.filename* exists.

filename
The artefact's filename.

justname
The filename without the extension and directory components.

```
>> Artefact('/dir/file.txt').justname
'file'
```

class plumbium.artefacts.**NiiGzImage**(*filename*, *exists=True*)
An artefact for `.nii.gz` images.

Parameters **filename** (*str*) – The filename of the artefact.

Keyword Arguments **exists** (*boolean*) – If true raise an exception if the file does not exist.

class plumbium.artefacts.**TextFile**(*filename*, *exists=True*)
An artefact for `.txt` files.

Parameters **filename** (*str*) – The filename of the artefact.

Keyword Arguments **exists** (*boolean*) – If true raise an exception if the file does not exist.

Module containing the get_environment function.

plumbium.environment.**get_environment**()
Obtain information about the executing environment.

Captures:

- installed Python packages using pip (if available),
- hostname
- uname
- environment variables

Returns a dict with the keys `python_packages`, `hostname`, `uname` and `environ`

Return type dict

Module containing functions for recording results to files and databases.

class plumbium.recorders.**CSVFile**(*path*, *values*)
Records results to a CSV file.

Parameters

- **path** (*str*) – The file to which results should be written
- **values** (*dict*) – a mapping from table columns to values

write(*results*)
Write results to the file specified.

Parameters **results** (*dict*) – A dictionary of results to record

---

Note: If the specified does not exist it will be created and a header will be written , otherwise the new result is appended.

---

class plumbium.recorders.**SQLDatabase**(*uri*, *table*, *values*)
Record results to a database supported by SQLAlchemy.

Parameters

- **uri** (*str*) – database server URI e.g. `mysql://username:password@localhost/dbname`
- **table** (*str*) – table name
- **values** (*dict*) – a mapping from database table columns to values

See also:

SQLAlchemy documentation

write(*results*)
Write the results to the database table specified at initialisation.

Parameters **results** (*dict*) – A dictionary of results to record

class plumbium.recorders.**MongoDB**(*uri*, *database*, *collection*)
Records results to a MongoDB database.

Parameters

- **uri** (*str*) – MongoDB server URI e.g. `mongodb://localhost:27017`
- **database** (*str*) – database name
- **collection** (*str*) – collection name

Note: Use of this class requires the installation of the pymongo module.

See also:

MongoDB tutorial

write(*results*)
Insert results into the database.

class plumbium.recorders.**StdOut**(*values*)
Print results to stdout.

Parameters **values** (*dict*) – key-value pairs to be printed

write(*results*)
Print the results to stdout.

class plumbium.recorders.**Slack**(*url*, *channel*, *values*)
Send a Slack notification when a pipeline completes.

Parameters

- **url** (*str*) – Slack Webhook URL
- **channel** (*str*) – The channel name to post to
- **values** – (dict): A mapping of result keys to report

Note: Use of this class requires the installation of the slackclient module.

---

write(*results*)
Send a message to Slack.

Parameters **results** (*dict*) – A dictionary of results to record

Exposes the CSVFile result recorder.

class plumbium.recorders.csvfile.**CSVFile**(*path*, *values*)
Records results to a CSV file.

Parameters

- **path** (*str*) – The file to which results should be written
- **values** (*dict*) – a mapping from table columns to values

write(*results*)
Write results to the file specified.

Parameters **results** (*dict*) – A dictionary of results to record

---

Note: If the specified does not exist it will be created and a header will be written , otherwise the new result is appended.

Exposes the MongoDB recorder class.

class plumbium.recorders.mongodb.**MongoDB**(*uri*, *database*, *collection*)
Records results to a MongoDB database.

Parameters

- **uri** (*str*) – MongoDB server URI e.g. `mongodb://localhost:27017`
- **database** (*str*) – database name
- **collection** (*str*) – collection name

Note: Use of this class requires the installation of the pymongo module.

See also:

MongoDB tutorial

write(*results*)
Insert results into the database.

Exposes the Slack result recorder.

class plumbium.recorders.slack.**Slack**(*url*, *channel*, *values*)
Send a Slack notification when a pipeline completes.

Parameters

- **url** (*str*) – Slack Webhook URL
- **channel** (*str*) – The channel name to post to
- **values** – (dict): A mapping of result keys to report

Note: Use of this class requires the installation of the slackclient module.

---

write(*results*)
Send a message to Slack.

Parameters **results** (*dict*) – A dictionary of results to record

Exposes the SQLDatabase result recorder.

class plumbium.recorders.sqldatabase.**SQLDatabase**(*uri*, *table*, *values*)
Record results to a database supported by SQLAlchemy.

Parameters

- **uri** (*str*) – database server URI e.g. mysql://username:password@localhost/dbname
- **table** (*str*) – table name
- **values** (*dict*) – a mapping from database table columns to values

See also:

SQLAlchemy documentation

write(*results*)
Write the results to the database table specified at initialisation.

Parameters **results** (*dict*) – A dictionary of results to record

Exposes the StdOut recorder.

class plumbium.recorders.stdout.**StdOut**(*values*)
Print results to stdout.

Parameters **values** (*dict*) – key-value pairs to be printed

write(*results*)
Print the results to stdout.

2.7 Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

J

M

N

O

P

R

S

T

W