


|  |                            |   |                       |
|--|----------------------------|---|-----------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                       |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b> |
|  |                            | Date: May 13, 2020                          | Page:<br><b>1</b>     |

false

|       |      |                                   |       |
|-------|------|-----------------------------------|-------|
| false | true | [placement=!ht]floatbartdanger    | false |
| true  | true | [placement=!ht]floatbarterror     | false |
| true  | true | [placement=!ht]floatbartwarning   | false |
| true  | true | [placement=!ht]floatbartcaution   | false |
| true  | true | [placement=!ht]floatbartattention | false |
| true  | true | [placement=!ht]floatbartnote      | false |
| true  | true | [placement=!ht]floatbartimportant | false |
| true  | true | [placement=!ht]floatbarthint      | false |
| true  | true | [placement=!ht]floatbarttip       | false |

Manuel Duarte and Grégoire Duvauchelle RPW ground segment software developers  
Xavier BONNIN RPW ground segment project manager  
Xavier BONNIN RPW ground segment project manager




# POPPy User Manual

ROC-TST-GSE-SUM-00035-LES


**Iss.02, Rev.00**


Release 02.00

| Prepared by                               | Date | Signature |
|---|------|-----------|
| <b>Manuel DUARTE</b><br>Software engineer |      |           |
| Approved by                               | Date | Signature |
| <b>Xavier BONNIN</b><br>Research engineer |      |           |
| Authorised by                             | Date | Signature |
| 1   |      |           |

|  |                   |   |                            |
|--|-------------------|---|----------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                            |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>      |
|  |                   | Date: May 13, 2020                          | falsefalsePage<br><b>i</b> |


falsetrue


|  |                   |   |                              |
|--|-------------------|---|------------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>ii</b> |


|  |                               |   |                    |                       |                    |                               |
|--|-------------------------------|---|--------------------|-----------------------|--------------------|-------------------------------|
|  | <p>POPPy User Manual</p>      | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>iii</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>iii</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>         |   |                    |                       |                    |                               |
| Date: May 13, 2020   | falsefalsePage:<br><b>iii</b> |   |                    |                       |                    |                               |


# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Pipeline description</b>                           | <b>3</b> |
| 1.1      | General . . . . .                                     | 3        |
| 1.1.1    | Scope of the document . . . . .                       | 3        |
| 1.1.2    | Applicable documents . . . . .                        | 3        |
| 1.1.3    | Reference documents . . . . .                         | 3        |
| 1.2      | Introduction . . . . .                                | 3        |
| 1.2.1    | About POPPy . . . . .                                 | 3        |
| 1.2.2    | Copyright, license and contact . . . . .              | 4        |
| 1.3      | Getting start with POPPy . . . . .                    | 4        |
| 1.3.1    | System requirements . . . . .                         | 4        |
| 1.3.2    | Prerequisites . . . . .                               | 4        |
| 1.3.3    | Setting up development environment . . . . .          | 5        |
| 1.3.4    | Create a pipeline . . . . .                           | 5        |
| 1.3.5    | Create a plugin . . . . .                             | 6        |
| 1.4      | Models and migrations . . . . .                       | 7        |
| 1.4.1    | Write models for the main database . . . . .          | 7        |
| 1.4.1.1  | Write models for a secondary database . . . . .       | 8        |
| 1.4.1.2  | Generate the migration . . . . .                      | 8        |
| 1.4.1.3  | Edit the migration . . . . .                          | 8        |
| 1.4.1.4  | Manage your migration branches . . . . .              | 9        |
| 1.4.1.5  | Run the migrations . . . . .                          | 9        |
| 1.4.2    | Use postgresql schemas . . . . .                      | 10       |
| 1.4.3    | Generate the documentation of your database . . . . . | 10       |
| 1.4.3.1  | The table dictionary . . . . .                        | 11       |
| 1.5      | Pipeline . . . . .                                    | 11       |
| 1.5.1    | Usage . . . . .                                       | 11       |
| 1.5.2    | Initialization . . . . .                              | 12       |
| 1.5.2.1  | Context setup . . . . .                               | 12       |
| 1.5.2.2  | Connector setup . . . . .                             | 13       |
| 1.5.2.3  | Dry run setup . . . . .                               | 13       |
| 1.5.3    | Task chain . . . . .                                  | 14       |
| 1.5.3.1  | Linking . . . . .                                     | 14       |
| 1.5.3.2  | Cutting chain . . . . .                               | 15       |
| 1.5.3.3  | Loop . . . . .  | 15       |
| 1.5.4    | Run . . . . .   | 16       |


|  |                      |   |                              |
|--|----------------------|---|------------------------------|
|  |                      | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
| <b>POPPy User Manual</b>   |                      | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                      | Date: May 13, 2020                          | falsefalsePage:<br><b>iv</b> |
|  | 1.5.4.1              | Topology generation . . . . .               | 16                           |
|  | 1.5.4.2              | Binding . . . . .                           | 17                           |
|  | 1.5.4.3              | Execution . . . . .                         | 17                           |
|  | 1.5.5                | API . . . . .                               | 17                           |
| 1.6  | Tasks . . . . .      |   | 17                           |
|  | 1.6.1                | Task creation . . . . .                     | 18                           |
|  | 1.6.1.1              | Using class . . . . .                       | 18                           |
|  | 1.6.1.2              | Using function . . . . .                    | 20                           |
|  | 1.6.1.3              | Using plugin . . . . .                      | 20                           |
|  | 1.6.2                | Communication . . . . .                     | 21                           |
|  | 1.6.2.1              | Dependency . . . . .                        | 21                           |
|  | 1.6.2.2              | Inputs/Outputs . . . . .                    | 22                           |
|  | 1.6.2.3              | Signals . . . . .                           | 23                           |
|  | 1.6.3                | API . . . . .                               | 24                           |
| 1.7  | Targets . . . . .    |   | 24                           |
|  | 1.7.1                | Target creation . . . . .                   | 24                           |
|  | 1.7.1.1              | Using class . . . . .                       | 24                           |
|  | 1.7.1.2              | Using task from plugin . . . . .            | 25                           |
|  | 1.7.2                | Usage . . . . .                             | 26                           |
|  | 1.7.3                | API . . . . .                               | 26                           |
| 1.8  | Commands . . . . .   |   | 26                           |
|  | 1.8.1                | Creating a new command . . . . .            | 26                           |
|  | 1.8.1.1              | Simple command . . . . .                    | 27                           |
|  | 1.8.1.2              | Add arguments . . . . .                     | 28                           |
|  | 1.8.2                | Hierarchy in commands . . . . .             | 30                           |
|  | 1.8.3                | Inheritance of parameters . . . . .         | 31                           |
|  | 1.8.3.1              | Simple use case . . . . .                   | 31                           |
|  | 1.8.3.2              | Advanced use case . . . . .                 | 32                           |
|  | 1.8.4                | API . . . . .                               | 33                           |
| 1.9  | Descriptor . . . . . |   | 34                           |
|  | 1.9.1                | Introduction . . . . .                      | 34                           |
|  | 1.9.2                | Descriptor interface . . . . .              | 34                           |
|  | 1.9.2.1              | Pipeline descriptor . . . . .               | 34                           |
|  | 1.9.2.1.1            | Identification . . . . .                    | 35                           |
|  | 1.9.2.1.2            | Release . . . . .                           | 35                           |
|  | 1.9.2.1.3            | Project . . . . .                           | 35                           |
|  | 1.9.2.1.4            | Databases . . . . .                         | 36                           |
|  | 1.9.2.1.5            | Calibration softwares . . . . .             | 36                           |
|  | 1.9.2.2              | Plugin descriptor . . . . .                 | 36                           |
|  | 1.9.2.2.1            | Identification . . . . .                    | 36                           |
|  | 1.9.2.2.2            | Release . . . . .                           | 37                           |
|  | 1.9.2.2.3            | Tasks . . . . .                             | 37                           |
|  | 1.9.2.2.4            | Inputs . . . . .                            | 38                           |
|  | 1.9.2.2.5            | Outputs . . . . .                           | 38                           |
|  | 1.9.2.2.6            | Example . . . . .                           | 38                           |


|  |                            |   |
|--|----------------------------|---|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>v</b></div> </div> |
| <b>2</b>   | <b>Developer Guide</b>     | <b>43</b>   |
| 2.1  | Workflow . . . . .         | 43  |
| 2.1.1  | API . . . . .              | 44  |
| <b>3</b>   | <b>TODO list</b>           | <b>47</b>   |
|  | <b>Python Module Index</b> | <b>49</b>   |

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>vi</b> |

|  |                   |   |                            |
|--|-------------------|---|----------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                            |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>      |
|  |                   | Date: May 13, 2020                          | falsefalsePage<br><b>1</b> |

falsetrue

|  |                   |   |                             |
|--|-------------------|---|-----------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                             |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>       |
|  |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>2</b> |

|   |                   |   |                             |
|---|-------------------|---|-----------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                             |
|   |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>       |
|   |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>3</b> |

# Chapter 1

## Pipeline description

### 1.1 General

#### 1.1.1 Scope of the document

This document is the POPPy framework user manual. It describes how to install and use the framework. It also provides tutorials for users.

#### 1.1.2 Applicable documents

| Mark | Reference/Issue/Revision | Title of the document |
|------|--------------------------|-----------------------|
|------|--------------------------|-----------------------|

#### 1.1.3 Reference documents

| Mark | Reference/Issue/Revision | Title of the document |
|------|--------------------------|-----------------------|
|------|--------------------------|-----------------------|

### 1.2 Introduction

#### 1.2.1 About POPPy

The Plugin-Oriented Pipeline for Python (POPPy) framework offers functionalities to develop, install and run in a standard way workflows. It is more particularly designed to work with data processing pipelines producing files.

POPPy supports basical features that are usually needed when building and executing pipelines, such as:

- Modular oriented architecture

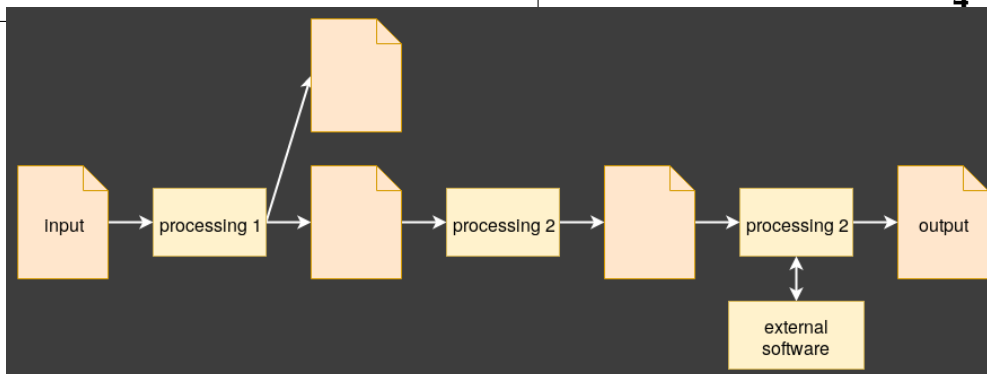


Fig. 1.1: Schema of a pipeline built with POPPy

- Jobs execution activity and status logging
- Centralized command line interface to execute batch jobs
- Input/output data handling traceability
- Standardized database communication

## 1.2.2 Copyright, license and contact

POPPy was first written in the framework of the RPW Operations Centre (ROC) project. The ROC is the main entity in charge of the ground segment of the Radio and Plasma Waves instrument (RPW) on-board the Solar Orbiter European space probe. Visit <http://rpw.lesia.obspm.fr/> for more details about Solar Orbiter, RPW and the ROC.

The POPPy framework is released under the TBD license.

The POPPy developer teams can be contacted via [roc.support@sympa.obspm.fr](mailto:roc.support@sympa.obspm.fr).

## 1.3 Getting start with POPPy


This section details how to install POPPy and use it to develop a pipeline.

### 1.3.1 System requirements

POPPy has been tested to work on Linux Debian operating system.

### 1.3.2 Prerequisites

Make sure that the following software set is installed on your system before deploying and using POPPy:

|  |                             |   |                    |                       |                    |                             |
|--|-----------------------------|---|--------------------|-----------------------|--------------------|-----------------------------|
|  | <h1>POPPy User Manual</h1>  | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>5</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>5</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>       |   |                    |                       |                    |                             |
| Date: May 13, 2020   | falsefalsePage:<br><b>5</b> |   |                    |                       |                    |                             |

- Python 3.6 or higher
- Git

Additionally a relational database managment system (RDMS) will be required to run a POPPy pipeline.

### 1.3.3 Setting up development environment

POPPy must be first installed to develop a pipeline.

It is strongly recommended to use POPPy into a Python's virtual environment (virtualenv) in order to avoid dependency conflicts. Since the version 3.5, the virtualenv mechanism is natively included in Python.

To create a virtualenv, open a terminal and enter:

```
$ python3 -m venv /path/to/myprojectvenv
```

Where /path/to/myprojectvenv is the path to the virtualenv's directory.

Then, to load enter the command:

```
$ source /path/to/myprojectvenv/bin/activate
```

For more details about the Python's virtual environments, please visit <https://docs.python.org/3/tutorial/venv.html>.

To install POPPy in the virtualenv, execute the three following commands successively:

```
$ pip install git+https://gitlab.obspm.fr/POPPY/POPPyCore.git@develop
↪#egg=poppy.core
$ pip install git+https://gitlab.obspm.fr/POPPY/POP.git@develop#egg=poppy.
↪pop
$ pip install git+https://gitlab.obspm.fr/POPPY/PIPER.git@develop
↪#egg=poppy.piper
```


The first command retrieves from the remote Git server and sets up the POPPy core library. The second and third commands retrieve and set up the POP and PIPER mandatory plugins.

### 1.3.4 Create a pipeline

You can generate a pipeline and all the boilerplate code needed to have a basic pipeline that uses the framework.

```
$ poppy create pipeline poppy_tuto
```

You will get a directory called *mypipeline/* in the current directory containing multiple files :

|  |                             |   |                    |                       |                    |                             |
|--|-----------------------------|---|--------------------|-----------------------|--------------------|-----------------------------|
|  | <h1>POPPy User Manual</h1>  | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>6</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>6</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>       |   |                    |                       |                    |                             |
| Date: May 13, 2020   | falsefalsePage:<br><b>6</b> |   |                    |                       |                    |                             |

```
poppy_tuto
- config.json
- descriptor.json
- lib
- manage.py
- requirements.txt
- settings.py
```

- `config.json` : Contains the output path of the pipeline, database credentials and address. This is the only file that should not be tracked by your vcs.
- `descriptor.json` : Provides metadata associated to the pipeline, the project and databases.
- `settings.py` : Contains the list of active plugins, a variable to the root directory and the identifier of the main database.
- `requirements.txt` : Contains the list of python libraries dependencies
- `lib/` : Contains eventual external libraries (in the case of the RPW pipeline, this directory contains nasa's CDF library and the Instrument Database)
- `manage.py` : The entry point of the pipeline.

### 1.3.5 Create a plugin

You can then create a plugin skeleton the same way we created the pipeline :

```
$ poppy create plugin guide.myplugin
```

Your plugin name must be of the form *namespace.pluginname*. It is once again a way to split the code in a meaningful way. To help you sort your code, create a directory called `plugins/` in root directory of your pipeline. However your plugins can be wherever you want.

You will see it has once again created a bunch of files prefilled with some usual code.


In order to use the namespace feature, the python code of your plugin must be located in the directory *plugin/namespace/plugin/* (see [PEP 420](#) for more information on namespaces)

In the plugin root directory there is :

- `setup.py` : it is a common python file, it allows you to install your python module using *pip*
- `system_reqs.ini` : you can put in this file eventual external libraries needed by the plugin

In the *myplugin/guide/myplugin/* you will find multiple python files. It is not mandatory for the POPPy framework to split your code into multiple files but it is simply a good practice, so POPPy assumes you would like to split your code and generates multiple files.

- `descriptor.json` : as for the pipeline, each plugin needs a descriptor file. In the case of plugins there are information about the plugin, the tasks it will perform and their targets (input and

|  |                            |   |                             |
|--|----------------------------|---|-----------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                             |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>       |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>7</b> |

output files).

- `commands.py` : in this file you should register the commands you want to call from the Command Line Interface (CLI).
- `tasks.py` : a file containing the tasks of your plugin. Usually those tasks are simply decorated python functions.
- `tests.py` : this prefilled file should encourage you to write unit/functional/end-to-end/whatever tests for your pipeline. The test procedure is integrated to the POPPy framework and wrapper classes and functions exists to help you.
- `models` : you will put in this directory all the database models corresponding to your plugin

## 1.4 Models and migrations

POPPy is built around [SQLAlchemy](#) , an Object-Relational Mapper (ORM) well known by Python developers.

The model integration needs some code to work, located in every `__init__.py` of the `models/` directory of each plugin. This code is automatically generated when you call the command `poppy create plugin`. If for some reason you do not have the code, you can find it in the POPPyCore source code at `POPPyCore/poppy/core/management/templates/plugin_template/plugin_namespace/plugin_name/models/__init__.tpl`

### 1.4.1 Write models for the main database

First, create a file in your plugin's *model* directory and add these imports:


```
from poppy.pop.models.non_null_column import NonNullColumn
from poppy.core.db.base import Base
```

- **NonNullColumn** is an improvement on SQLAlchemy's `Column` class. It is non null by default and allows us to autogenerate the documentation corresponding to the model we are writing.
- **Base** is the base class for any SQLAlchemy model and the one located in `poppy.core.db.base` is the one corresponding to the main database.

Then, you can create your model class inheriting from the base class, as you would any SQLAlchemy model:

```
class Dictionary(Base):

    id_dictionary = NonNullColumn(INTEGER(), primary_key=True, unique=True)
    word = NonNullColumn(String(16), descr='The word', unique=True,
↪comment='Must be an english word')
```

|  |                            |   |
|--|----------------------------|---|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>8</b></div> </div> |
|--|----------------------------|---|

```
__tablename__ = 'dictionary'
```

In this class you will define every column of your table including their type (either general or dialect specific types). If you want to generate your database documentation automatically (see chapter TBC), you should use poppy's custom column class `NotNullColumn` instead of `sqlalchemy.Column`.

The fields `description` and `comment` are used to generate the documentation.

```
true
```

### Note false

Do not use double quotes in your description or comment, but only escaped single quotes. ■

```
true
```

## 1.4.1.1 Write models for a secondary database

You might want to use SQLAlchemy to interface an already existing secondary database. In this case, you don't need to detail every column of the table. You only need to create a class that inherits from `Base` and `DeferredReflection`, and give the table name. You don't even need a primary key.

```
from sqlalchemy.ext.declarative import DeferredReflection

class PacketType(DeferredReflection, Base):
    __tablename__ = "packettype"
```

## 1.4.1.2 Generate the migration

The migration generation is integrated in the POPPy framework through calls to `alembic`. It is able to generate distinct migrations for every plugin.


```
python manage.py db makemigrations tuto.texter
```

You can use any argument of the `alembic revision` command as follows

```
python manage.py db makemigrations tuto.texter --message='First tuto.
↳texter migration' --depends_on='poppy.pop' --head=poppy.pop
```

## 1.4.1.3 Edit the migration

Alembic migration generation is not perfect and you have to manually review and correct the migrations that alembic produces. Furthermore, alembic is not yet fully integrated to the POPPy framework and

|  |                            |   |
|--|----------------------------|---|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>9</b></div> </div> |
|--|----------------------------|---|

some modifications has to be done in order for the migration to be registered by the framework. See (POPPy improvements) chapter.

First add this import

```
from poppy.pop.alembic.helpers import create_table, execute, user, drop_
↪table
```

Then, replace every `op.create_table` with the poppy wrapper function `create_table`. It is needed to grant access to the pipeline user and log what is happening. Do the same with `drop_table`. At this point you can also edit the alembic revision identifiers, for example add a branch label for convenience and the migration this one depends on (remember you can do all of this with the command arguments)

Finally, you should check the migration for any mistake that alembic may have made. See [this alembic documentation page](#) to know what alembic can and cannot detect.

true true

#### Note false true 2

- Alembic does not take in charge the creation of the Postgres database schemas and the user grant. Make sure that these commands have been added correctly in the migration script generated by Alembic.
- Make sure that the `branch_labels` and `depends_on` keywords are well defined in the migration script before running the migration.

true true

### 1.4.1.4 Manage your migration branches

You can manage your branches as explained in the [alembic documentation about branches](#)


You can call alembic commands through the pipeline with this command, so you don't have to worry about users and databases or even alembic configuration:

```
python manage.py db alembic ...
```

### 1.4.1.5 Run the migrations

You can either run migrations one by one by giving their branch label:

```
$ python manage.py db upgrade poppy.pop
$ python manage.py db upgrade tuto.texter
```

|   |                              |  |                    |                       |                    |                              |
|---|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>10</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>10</b> |
| Issue<br><b>02</b>  | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020  | falsefalsePage:<br><b>10</b> |  |                    |                       |                    |                              |

or all in one go with:

```
$ python manage.py db upgrade heads
```

## 1.4.2 Use postgresql schemas

If you want to use postgres schemas, you need to specify it in the models, in the `__table_args__` field:

```
class Dictionary(Base):

...

__table_args__ = {
    'schema': 'my_schema'
}
```

Then, when editing the first migration you need to manually add the creation and deletion of the schema :

```
from poppy.pop.alembic.helpers import create_schema, execute, drop_schema

def upgrade():
    create_schema('my_schema')

...

def downgrade():
    drop_schema('my_schema', cascade=True)
```

## 1.4.3 Generate the documentation of your database


Using this command :

```
poppy gen_doc your.plugin
```

You can generate a reST document containing the description of every database table of the given plugin. There is one document per plugin. The document is generated Here is an example of rst code generated and what it looks like once compiled.

```
The table dictionary
~~~~~
The "dictionary" table contains the dictionary

.. csv-table:: dictionary
```

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>11</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>11</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>11</b> |  |                    |                       |                    |                              |

```

:header: "Column name", "Data type", "Description", "Priority", "Comment"
↪"

    "id_dictionary", "INTEGER", "Primary key", "PK", ""
    "identifier", "INTEGER", "Identifier of the word", "NN", "Must be_
↪unique. "
    "word", "VARCHAR(16)", "The word", "NN", "Must be unique. Must be an_
↪english word."

The tuple of columns (identifier,word) must be unique.

```

### 1.4.3.1 The table dictionary

The “dictionary” table contains the dictionary

Tab. 1.1: dictionary

| Column name   | Data type    | Description            | Prior-ity | Comment                                  |
|---------------|--------------|------------------------|-----------|--|
| id_dictionary | INTEGER      | Primary key            | PK        |  |
| identifier    | INTEGER      | Identifier of the word | NN        | Must be unique.                          |
| word          | VAR-CHAR(16) | The word               | NN        | Must be unique. Must be an english word. |

The tuple of columns (identifier,word) must be unique.

truetrue

#### Note falsetrue3


The code responsible for the documentation generation is located in poppy.core.management.commands in the function `gen_doc()` ■

truetrue

## 1.5 Pipeline

### 1.5.1 Usage

To launch the pipeline:

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>12</b></div> </div> |
|--|----------------------------|--|

```
$ python manage.py
```

It should show the help message of the pipeline. A description of the available options is done, and also for sub-commands provided by the pipeline or its plugins.

Since information provided by plugins are displayed when launching the pipeline, some jobs have already done by the module `poppy.pop`.

The workflow is as follows and is described more thoroughly in the developer guide:

1. Creation of the pipeline object. It uses the arguments provided (usually from the CLI) in order to initialize correctly the databases.
2. Link of the tasks (chain) with the pipeline.
3. Run the pipeline with the provided tasks.

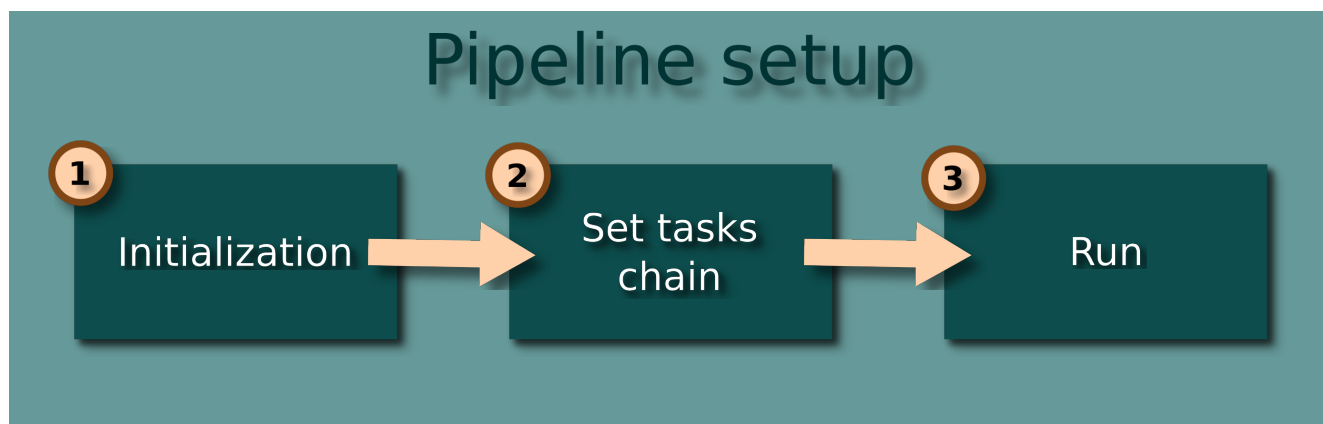


Fig. 1.2: Steps of the pipeline setup.

## 1.5.2 Initialization


The initialization of the pipeline object is divided in several parts. They are detailed below.

### 1.5.2.1 Context setup

The context setup is an important part of the pipeline. This is an attribute containing all the variable necessary to the pipeline to setup its environment. But it is also a way for the tasks to share information between them across all the chain.

This attribute is called `poppy.pop.pop.Pop.properties`, and is an instance of `poppy.core.properties.Properties`, a dictionary like class whose attributes can also be accessed as dictionary key. The existence of an attribute inside this context can be tested easily with a simple `in` operator.

All the arguments `args` transmitted to the pipeline are set on the context (properties) of the pipeline, thus allowing any connected task to use settings from the environment or the user. This is done through:

|   |                              |  |                    |                       |                    |                              |
|---|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>13</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>13</b> |
| Issue<br><b>02</b>  | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020  | falsefalsePage:<br><b>13</b> |  |                    |                       |                    |                              |

```
self.properties = Properties()
self.properties += vars(args)
```

where `vars` simply takes the attributes, store them into dictionary and add them to the context.

### 1.5.2.2 Connector setup

From the information provided by the arguments now stored into the context, the pipeline can setup the connectors. A connector permits to link a database with an identifier to an unique connection object in the pipeline. This connector will remain the same in the code, but another database can be linked to it later if necessary.

The pipeline's context is set as an attribute of the connector after its creation, allowing the connector to shortcut the pipeline when necessary, in order to access the information in the context for example.

The function `poppy.core.db.database.link_databases()` loop over defined databases, create the connector if not already created and set the linked database as an attribute to the connector.


```
"databases": [
    {
        "identifier": "MAIN-DB",
        "connector": "poppy.pop.roc_connector.ROC",
        "login_info": {
            ...
        }
    }
]
```

- `identifier` is the identifier of the database that will be linked to the connector and that will be used to make the necessary connections along the program. It is also the identifier that is used as reference in the code to get the associated connector.
- `connector` is optional. If present, the class in the provided module path will be used to construct the connector, else the default `poppy.db.connector.Connector` is used. It can be useful to add other behaviour to a given connector.

### 1.5.2.3 Dry run setup

The dry run object `poppy.core.db.dry_runner.DryRunner`, a singleton in the program, is referenced by the pipeline to enable/disable the dry run mode in function of the settings used by the user.

The dry run object allows to enable/disable some functions, methods in the code at runtime, simply by decorating them. For example, you can decorate a method with the dry run object to not execute this method if the dry run mode is activated. This allows, for example, to not write things into the ROC database if this mode is activated, while keeping the other features of the POPPy framework intact.

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>14</b></div> </div> |
|--|----------------------------|--|

The state of the dry run mode is setup according to the status of the `--dry-run` argument of `python manage.py`.

```
self.dry_runner = DryRunner()
self.dry_run = args.dry_run
```

Here, the value in the CLI is present in `args` and the `dry_run` attribute set the state of the dry run mode through the setter.

```
@property
def dry_run(self):
    return self._dry_run

@dry_run.setter
def dry_run(self, dry_run):
    self._dry_run = dry_run
    if dry_run:
        self.dry_runner.activate()
    else:
        self.dry_runner.deactivate()
```

## 1.5.3 Task chain

### 1.5.3.1 Linking

Linking a task chain to the pipeline instance is simple. Just create a pipe link between a task and the pipeline. Then the task can be linked (or already linked) to other tasks.

The pipeline will keep a reference to this task called the entry point to be able to walk through the graph formed by the tasks to run.


At each linking, a flag is set to indicate that the graph of tasks will have to be regenerated before the execution of the pipeline.

true

#### Important false

In fact, the entry point task is stored into the pipeline but will not be used. A flag is set to indicate that the chain has changed and need an update. This the information from the `poppy.pop.pop.Pop.start` that gives really an entry point. ■

true

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>15</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>15</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>15</b> |  |                    |                       |                    |                              |

### 1.5.3.2 Cutting chain

If a chain of tasks is already existing, and just a small part of it must be executed, it can be useful to only run this part, without the extra-tasks. This is why `poppy.pop.pop.Pop.start` and `poppy.pop.pop.Pop.end` attributes are existing. The `poppy.pop.pop.Pop.start` attribute must be set to indicate the starting point for the chain of tasks to execute. The `poppy.pop.pop.Pop.end` attribute is not mandatory. It can be set to the end of the task if all do not have to be done inside the chain.

### 1.5.3.3 Loop

A loop feature gives the possibility to rerun according to an iterator a part of the tasks chain. A loop can be created by calling the `poppy.pop.pop.Pop.loop()` method with the starting task, ending task and the iterator that will be iterated to get the step of the loop.

An instance of `poppy.pop.loop.Loop` will be created, that will override the settings for ancestors and descendants of the start and end tasks provided in arguments. This instance will connect to the `poppy.pop.task.Task.errorred` signal, emitted each time that an error occurred in a task inside the loop chain, to handle correctly an error while in the loop.

truetrue

#### Note falsetrue4

It will not connect to task outside the path(s) between the start and end task of the loop. ■

truetrue

Start and end tasks will also be monkey patched appropriately to handle errors occurring on these tasks and also the end of the loop. The following flowchart in Fig. 1.3 gives an idea on what have to be done on each signal emitted by tasks.

The flowchart takes the example of 6 tasks A, B, C, D, E, F in this order of execution, whose tasks B, C, D, E are inside a loop. When setting the start task, the loop instance will connect to the `started` signal of task B. The same is done for the `ended` signal of end task E. All others tasks in the path(s) between B and E are connected to their `errorred` signal, emitted when an error happened on the task.

If an error happens on task C or the end of an iteration is reached in E, the connected slot of the `loop^instance` is called. The following steps are executed:

1. check if the loop can continue or not (`StopIteration` exception) from the given iterable.
2. **YES:** monkey patch descendants of the end/error task to point to the start task.
3. Reset status of tasks inside the loop.
4. keep a reference to the end/error task.
5. **NO:** there is no more iteration to do for the loop. Disconnect start task from the slot of the loop instance.

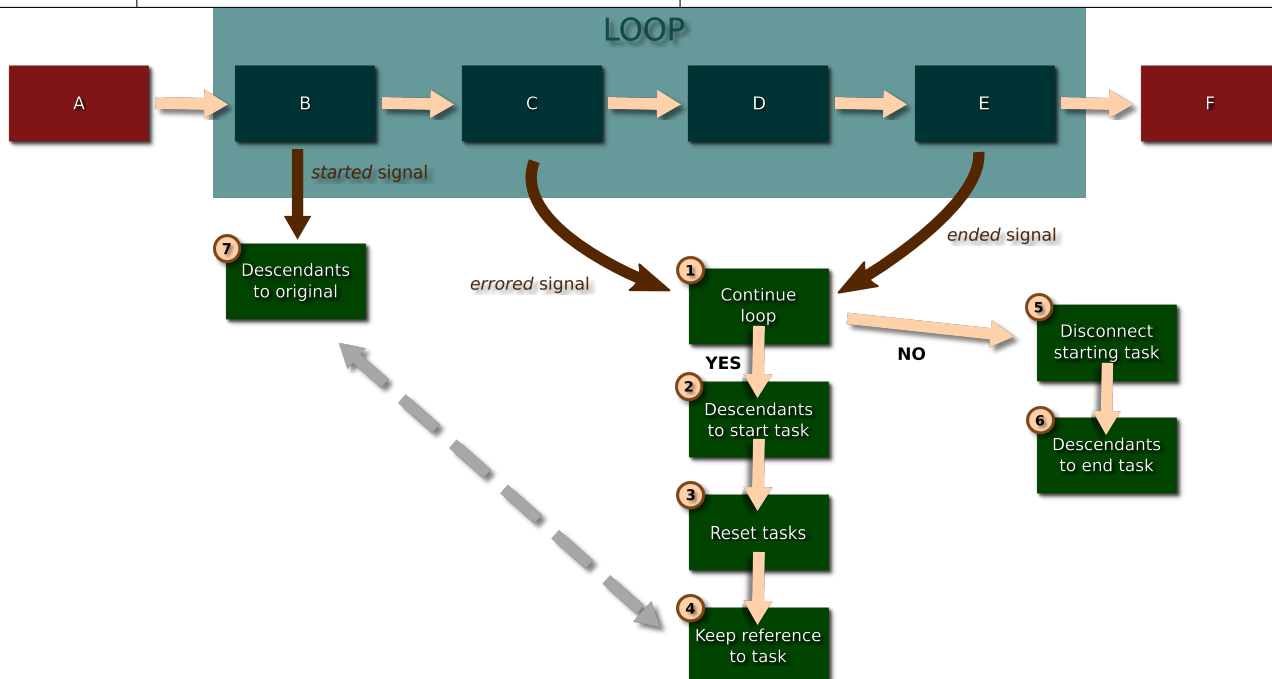


Fig. 1.3: Flowchart for the loop of the pipeline for a chain of 6 tasks. Tasks B, C, D, E are placed inside a loop. The loop instance changes the descendants of tasks dynamically in function of the status of tasks inside the loop.

6. also changes the descendants of the task to the one of the end task. Thus, following tasks not in a loop are executed as usual.
7. At next iteration, if a task is found for the reference task, its descendants are again set to the original method to get them.


With this process of dynamic change of the topology of the pipeline inside a loop, no need to integrate it in the main pipeline process. The pipeline works as usual, it is just an other instances that take care of what is happening inside the task chain.

## 1.5.4 Run

### 1.5.4.1 Topology generation

The first thing to do before running the pipeline is to create the dependency graph of the task chain. From the starting task, the pipeline goes through the tasks in chain and create the graph of the tasks, that will be used to find paths between tasks. At the same time, the pipeline adds its reference into each task, allowing them to access to the context, and any other data provided by the pipeline.

If the topology has already been created (the flag is set), the topology is not created again.

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>17</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>17</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>17</b> |  |                    |                       |                    |                              |

### 1.5.4.2 Binding

Then, the pipeline binds the ROC connector. Since the ORM uses the reflection system to create the class doing the mapping with the database, we need to indicate to `sqlalchemy` at which moment to check in the database for creating the mapping. This is done at the `poppy.core.db.connector.Connector.bind()` method. This will try to get the linked database object, create the mapping classes with the databases with reflection if not already done, and then attempt a connection with the database.

The pipeline does this automatically at startup to not let the user do it, avoiding incomprehensible error message if not binded. The pipeline does it only for its own connector. Other connectors are not binded by the ROC pipeline.

### 1.5.4.3 Execution

The execution of the pipeline simply consists in running all dependencies of a task before itself and its children. This can normally be done recursively. But since a loop feature has been introduced, this can create situations where the maximal recursion depth limit of python has been reached. This way another approach with queue has been implemented.

A `while` loop is executed until the queue becomes empty. A task is *popped* from the queue. If the task is already completed or failed, the next iteration is performed, and a new task is *popped*. If task dependencies (parents) are not already executed, they are added into the queue. If not all its dependencies are completed, the loop continues.

If all dependencies are done, the task itself is executed. Then all its children are added into the queue to be run themselves.

## 1.5.5 API


## 1.6 Tasks

Tasks are elemental bricks of the pipeline. The pipeline is composed of a succession of tasks, linked together by dependencies to each other tasks. Thus, the pipeline is just the topology resulting of the dependencies of tasks.

A task is just a succession of instructions to run, with its inputs and outputs provided and/or used by the other tasks. A task can be launched if its dependencies are complete and if the required inputs presents.

Tasks can communicate between them by sharing the inputs and outputs they use through the pipeline, and/or via properties stored also by the pipeline. So, the pipeline can be seen as a manager of tasks, regulating the communication between them and checking that they are completed before starting a new task with dependencies.

There are three ways of creating a task, described in sections below.

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>18</b></div> </div> |
|--|----------------------------|--|

## 1.6.1 Task creation

### 1.6.1.1 Using class

To create a task, you can simply derive from the base class `poppy.pop.Task`. At the instantiation, the task needs some information on the kind of jobs it will perform, such as the software related to the task, the category of the task and a description to be able to have information on the task in the database simply by looking at it.

To be executable, the task must provide a `run()` method, taking no arguments. This the *main* of the task. The work to perform must be done in this method.

For example, to create a task that displays *Hello world!*:

```
from poppy.pop import Task


class HelloTask(Task):
    """
    Example task printing a message on the terminal.
    """
    def run(self):
        """
        The method launched by the pipeline to start the task.
        """
        print("Hello World!")
```

Then to instantiate the task, you will have to do:

```
task = HelloTask("Software category", "Task category", "A description")
```

The first argument is the software category, in other words the software to which the task is linked (it must be one valid for the current version of the POPPy framework, available in the descriptor file, see TODO for details). The second argument is the category of the task, it must be one accepted by the ROC database.

true true

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>19</b></div> </div> |
|--|----------------------------|--|

## Note falsetrue5

If you have to create multiple similar tasks, it can be constraining to always specify the same arguments at their creation. You can also create a new task category and specify the arguments once at the instantiation by overriding the `__init__()` method.

```
class HelloTask(Task):
    """
    Example task printing a message on the terminal.
    """
    def __init__(self):
        """
        Override the parameters to put at each instantiation.
        """
        super(HelloTask, self).__init__(
            "Software category",
            "Task category",
            "A description",
        )

    def run(self):
        """
        The method launched by the pipeline to start the task.
        """
        print("Hello World!")
```

and then you can simply do:

```
task = HelloTask()
```

for each task of this kind that you want to create. ■


truetrue

truetrue

## Warning falsetrue1

You should note that the created task in the example above is not already linked to the pipeline. This will be described in the dedicated section. ■

truetrue

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>20</b></div> </div> |
|--|----------------------------|--|

### 1.6.1.2 Using function

Sometimes, writing a class for each task can be a little annoying, and writing a function faster. So, the Task provides a class method to be able to decorate a function and transform it into a task.

The last example can be rewritten:

```
@Task.as_task
def HelloTask(task):
    print("Hello World!")
```

Much more compact! But you still have to pass mandatory parameters at the instantiation of the task. So you can combine the best of both worlds, by declaring a task with fixed parameters, and use it to decorate many other functions to create other tasks.

```
class HelloTask(Task):
    """
    Example task printing a message on the terminal.
    """
    def __init__(self):
        """
        Override the parameters to put at each instantiation.
        """
        super(HelloTask, self).__init__(
            "Software category",
            "Task category",
            "A description",
        )

@HelloTask.as_task
def HelloFunction(task):
    print("Hello World!")


# instantiation of the task
task = HelloFunction()
```

### 1.6.1.3 Using plugin

If a plugin following the pipeline interface is defined and activated, it can be used to define a task. For example, if in the descriptor of the plugin (see *Plugin descriptor*) is defined a task called `hello_world` with the good software category, description, etc, you can simply create a task from this definition. Let assume that the plugin is called `talker`:

```
from poppy.pop.plugins import Plugin

# create the class of the task from the definitions of the pipeline
HelloTask = Plugin.manager["talker"].task("hello_world")
```

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>21</b> |

```
# instantiation of the task
task = HelloTask()
```

```
true
```

### Note false

Tasks through plugins contains also extended functionalities allowing for example to define *targets* simply by name from their definition in the descriptor. Refer to the section *Targets* for details and description of functionalities. ■

```
true
```

## 1.6.2 Communication

Communication term is used to refer to the ways that a task has to share information with other tasks or the pipeline.

### 1.6.2.1 Dependency

Expressing the dependency between several tasks is simple as writing Unix pipes. First declare some tasks.


```
@NoParametersTask.as_task
def taskA(task):
    print("Task A")

@NoParametersTask.as_task
def taskB(task):
    print("Task B")

@NoParametersTask.as_task
def taskC(task):
    print("Task C")

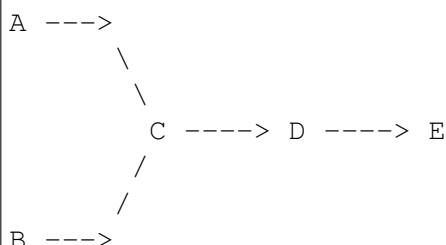
@NoParametersTask.as_task
def taskD(task):
    print("Task D")

@NoParametersTask.as_task
def taskE(task):
    print("Task E")
```

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>22</b></div> </div> |
|--|----------------------------|--|

with `NoParametersTask` a class task where the mandatory parameters for the task instantiation are already set.

To express the following dependency:



where **E** depends on **D**, which depends on **C**, which itself depends on **A** and **B**, you can write with tasks:

```

# create the C task
c = taskC()

# create the other tasks and set the topology as in the schema
taskA() | c | taskD() | taskE()

# express here the second branch of the graph of dependencies
taskB() | c

```

### 1.6.2.2 Inputs/Outputs

The pipeline needs to know what are the inputs and outputs of the task to check for their existence before and after starting it. The outputs of a task should always be created and existing before launching its children tasks. Same for the inputs.

For this, the `Task` gives two methods `input()` and `output()` returning the list of the name of attributes of the `properties` attribute of the pipeline where are stored the targets of the task. Targets are just the names of the wrapper of the inputs/outputs of the task, used to trace changes in their status and report them in the ROC database. More details on targets at *Targets*.


The pipeline uses these names to check there existence in the `properties` attribute, which is a container whose attributes are accessible as in a dictionary or as in a class instance, and that can be set in the same way. This is useful to refer to the attributes as names but hide this behaviour to the user. This is the `properties` that is used to share data and information between tasks.

In the case of tasks created through a function, the decorator gives the possibility to specify those lists:

```

@NoParametersTask.as_task(
    inputs=["file1", "file2"],
    outputs=["file1", "file3"]
)
def some_task(task):
    # do some work with files...

```

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>23</b> |

```
print("I'm working!")

# instantiate the task
task = some_task()
```

### 1.6.2.3 Signals

A task emits some signals on which slots can be connected to be called when the signal is triggered.

An interface is provided in order to have the pipeline being able to deal with the changes of state of the task. Calling this methods on the task instance will let the possibility to emit the signal without knowledge of the signature of the signal.

The list of signals is:

- **changed:** Emitted when the representation of the task in the ROC database as changed.
- **created:** Emitted when the representation of the task in the ROC database as been created.
- **started:** Emitted when the task started Usually resulting of the call of `start()` method on the task instance.
- **ended:** Emitted when the task stopped. Usually resulting of the call of `stop()` method on the task instance.
- **reseted:** Emitted when the internal states of the task instance have been reseted to their default values. Usually resulting of the call of `reset()` method on the task instance.
- **errored:** Emitted when an error occurred when the task was running or not. Usually resulting of the call of `error()` method on the task instance.


For example, to call a function each time a task as an error, you can do:

```
>>> def call_on_error(task):
>>>     """
>>>     Called when an error occurred in the task on which it is connected.
>>>     """
>>>     print("{0} have an error!".format(task))

>>> # connect the slot to the signal
>>> task.errored.connect(call_on_error)

>>> # say an error occurred
>>> task.error()
Task have an error
```

If you want to disconnect from the signal, simply do:

|   |                              |  |                    |                       |                    |                              |
|---|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>24</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>24</b> |
| Issue<br><b>02</b>  | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020  | falsefalsePage:<br><b>24</b> |  |                    |                       |                    |                              |

>>> task.errorored.disconnect(call\_on\_error)

truetrue

**Warning falsetrue2**

Slots are registered with weak references. It means that the slot you are connecting to a signal doesn't have its reference counter incremented, and thus the garbage collector will remove it if it is not referenced somewhere. Be sure to have a reference of the slot somewhere if you want to keep having the signal calling it!

truetrue

1.6.3 API

**exception** poppy.core.task.**TaskError**  
 Bases: Exception  
 Error associated to tasks.  
**\_\_weakref\_\_**  
 list of weak references to the object (if defined)


1.7 Targets

Targets are the way used by the pipeline to know if the expected Inputs/Outputs (I/O) of a task are existing/produced. A target should contain all the information necessary to the pipeline to make its decision on running or not the next step of the chain of treatment.  
 An instance of a target is fully identified by the its identifier, its version and the name of the file. Thus, the target can be seen as a category/group of file with a *physical* representation of them.

1.7.1 Target creation

1.7.1.1 Using class

A target instance can be created simply by instantiating a poppy.pop.target.Target class. The mandatory parameters have to be given at the instantiation. For example, a target for the dataset ROC-SGSE\_LZ\_MEB-SGSE-TEST-LOG in version 01, and a *physical* file /path/to/dataset/file:

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>25</b></div> </div> |
|--|----------------------------|--|

```
from poppy.pop import Target

target = Target(
    "/path/to/dataset/file", # filename of the dataset
    "ROC-SGSE_LZ_MEB-SGSE-TEST-LOG", # identifier for the target
    "01", # version of the target
)
```

This results in a target instance that can be used and shared across the pipeline, through the context. Placing it in the context allows the pipeline to easily find targets used as I/O for tasks. See `target_usage` for details.

### 1.7.1.2 Using task from plugin

A target can also be created from a task instance created from a plugin. The plugin contains all necessary information for the target creation, allowing an easy and maintainable way of referencing and creating target instances. See *Tasks*.

If for example the target is named `xml_test_log` in the descriptor file of a plugin called `DARE-SGSE`, a target instance can easily be created with (assuming a task named `to_xml_file`):

```
from poppy.pop.plugins import Plugin

# create the class for the task
Task = Plugin.manager["DARE-SGSE"].task("to_xml_file")


# create a fake functionality for the task
@Task.as_task
def run(task):
    """
    task is an instance of the task to_xml_file, created from a plugin. It
    can be used to create a target from informations provided in the
    descriptor.
    """
    # first get the class of the target
    Target = task.target("xml_test_log")

    # create the instance with a path to the file
    target = Target("/path/to/target")
```

true

#### **Danger**

If it is an input target, `input_target` must be used instead of `target` as method on the task instance.

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>26</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>26</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>26</b> |  |                    |                       |                    |                              |

truetrue

## 1.7.2 Usage

All status changes of a target must be done inside its context. It allows the target to intercept any problem occurring while generating a file in input or output, and to gives the good error information to the POPPy framework.

For example, to use the context of the target and open the file inside it:

```
# use the context of the target
with target.activate():
    # open the file
    with target.open("r") as f:
        # do things with the file
```

This example is simple, but shows how the target handles all the job for the user. The status of the target is automatically updated according to each step. In case of an error inside this context, the status of the target is set accordingly to `poppy.pop.target.Target.error()` and reported into the database.

At startup, it is marked as `poppy.pop.target.Target.ok()` and `poppy.pop.target.Target.pending()`.

If while treating the file, the target is detected as empty, the **:exception: 'poppy.pop.target.Target.TargetEmpty'** must be raised, handled by the context to mark the target as empty. It is used by the pipeline to know that a file has been *treated* correctly, but that it is not existing, and this is normal.

The available status of a target are listed in *API*.


## 1.7.3 API

# 1.8 Commands

New commands can be easily added to the pipeline through the command framework.

## 1.8.1 Creating a new command

Commands are managed by the pipeline, and those the module containing the framework for the pipeline is defined in `poppy.pop.command`. To create a new command, simply derive a class from `Command`.

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>27</b></div> </div> |
|--|----------------------------|--|

### 1.8.1.1 Simple command

For example, we will create a command to display informations on the pipeline, such as who wrote the pipeline.

```
from poppy.pop import Command

class WhoCommand(Command):
    """
    Define a command displaying informations on the pipeline author.
    """
    # a name to reference the command
    __command__ = "who_command"

    # the name of the command used
    __command_name__ = "who"

    # the parent of the command
    __parent__ = "master"

    # the help message displayed to the user
    __help__ = "Show the author of the pipeline"

    def __call__(self, args):
        print("Super Mario, pipeline expert")
```


That's all!

Calling the pipeline with the *who* sub-command will display the name of the author. Now the details of the parameters.

The `WhoCommand` inherits the `Command`. By doing that, the `WhoCommand` is automatically registered by the framework, and the command is made available to the pipeline. Some attributes allows to perform some selection and modify the behaviour of the command.

- `WhoCommand.__command__` is used to give a reference name to the command. If the command associated to the `WhoCommand` needs to be referenced somewhere, this is the name defined by `WhoCommand.__command__` that will be used.
- `WhoCommand.__command_name__` is the name of the command. When invoking the sub-command, this is the `WhoCommand.__command_name__` that will be used.
- `WhoCommand.__parent__` is the reference to the parent sub-command. If you want that your command to be used as a sub-command of another command, simply add the reference name of another command inside this variable.
- `WhoCommand.__help__` contains the message to display to the user when invoking the help.

true true

|  |                   |   |                              |
|--|-------------------|---|------------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>28</b> |

## Warning falsetrue3

In order to the framework be able to discover the command just defined, you should be sure of two things:

- the class of your command inherits the `Command` class
- the class you defined has been imported somewhere before invoking the command. Typically an import has been done somewhere, happening before the execution of the function in the scripts directory.

truetrue

### 1.8.1.2 Add arguments

Let's say you are an exigent user and you want to have the possibility to select the format for displaying the name of the author. A way to do that is to create arguments that will be used in the command handler. For this, you just have to define a method to add argument to the created parser for the command, parser created by `argparse`.

So we add an option to the `who` command to display only the information on the author name, and a second one to display only the short description. Rewriting `WhoCommand`:


```
class WhoCommand(Command) :
    """
    Define a command displaying informations on the pipeline author.
    """
    # a name to reference the command
    __command__ = "who_command"

    # the name of the command used
    __command_name__ = "who"

    # the parent of the command
    __parent__ = "master"

    # the help message displayed to the user
    __help__ = "Show the author of the pipeline"

    def add_arguments(self, parser):
        """
        Add arguments to the parser associated to the command.
        """
        # add option for displaying only the author name
        parser.add_argument(
```

|  |                            |   |                 |
|--|----------------------------|---|-----------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                 |
|  |                            | Issue                                       | Revision        |
|  |                            | <b>02</b>                                   | <b>00</b>       |
|  |                            | Date: May 13, 2020                          | falsefalsePage: |
|  |                            |   | <b>29</b>       |

```

        "--author",
        help="Show only author name",
        action="store_true",
    )

    # same but only for the description
    parser.add_argument(
        "--description",
        help="Show only the description of the author",
        action="store_true",
    )

def __call__(self, args):
    if args.author:
        print("Super Mario")
    elif args.description:
        print("pipeline expert")
    else:
        print("Super Mario, pipeline expert")

```

Now doing:

```

$ pipeline who
Super Mario, pipeline expert

$ pipeline who --author
Super Mario

$ pipeline who --description
pipeline expert

```


If you want to specify to the user that the two options are mutually exclusives, you can the arguments inside a group. Simply modify `add_arguments()` to:

```

def add_arguments(self, parser):
    """
    Add arguments to the parser associated to the command.
    """
    # create a group
    group = parser.add_mutually_exclusive_group()

    # add option for displaying only the author name in the group
    group.add_argument(
        "--author",
        help="Show only author name",
        action="store_true",
    )

```

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>30</b> |

```
# same but only for the description
group.add_argument(
    "--description",
    help="Show only the description of the author",
    action="store_true",
)
```

Now, an error is displayed to the user if both options are given at the same time.

### See also:

Details on all the large possibilities on how to add arguments and actions that can be taken with them are given on `argparse`.

## 1.8.2 Hierarchy in commands

Now, you have a working pipeline, with several author contributions, and you want to show a list of them through the command line interface. This is clearly related to the *who* command, but you do not want to simply add arguments, since you may want to add arguments to make some filtration on the list you want to show. For this, you will need to add a subcommand to the *who* command. This can be easily achieved by creating a new `Command`.


```
class WhoListCommand(Command):
    """
    Command displaying the list of authors and contributors.
    """
    # reference for the command
    __command__ = "who_list_command"

    # command used in cli
    __command_name__ = "list"

    # here specify that the parent command is the who one
    __parent__ = "who_command"

    # help message to display for this command
    __help__ = "Show a list of contributors to the pipeline"

    def __call__(self, args):
        """
        With the list of authors taken from somewhere, display it when
        invoked as a command.
        """
        # simple message
        print("Contributors:")
```

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>31</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>31</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>31</b> |  |                    |                       |                    |                              |

```
# print the list of authors
print(", ".join(get_authors_list()))
```

The command is used like this:

```
$ pipeline who list
Contributors:
Super Mario, Luigi, Peach
```

This is simply the `__parent__` that will order the hierarchy of commands between them. You can add any arguments as needed to this command as described above. The arguments and options will be associated to this command only.

true true

### Note false true 7

The framework is very flexible and allows quick development of new commands and change in the hierarchy. Let's say that you do not want to use the `list` command with the `who` command anymore, and want it to be a "root" command, independent of the `who` one. The only thing you have to do is changing `__parent__` to `__master__` and rename the command if you wish with the `__command_name__` attribute. Resulting in a calling interface like this:

```
$ pipeline list
Contributors:
Super Mario, Luigi, Peach
```

true true

## 1.8.3 Inheritance of parameters


### 1.8.3.1 Simple use case

A command that you define can inherit the commands of a parent command if you specify it. By default, arguments of a command must be placed between the command and its subcommand, else the arguments will not be recognized.

```
$ pipeline command1 --arg_command1 command2
```

This command is valid. But the following not:

```
$ pipeline command1 command2 --arg_command1
```

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>32</b></div> </div> |
|--|----------------------------|--|

while it seems natural to do it this way if for example, **-arg\_command1** is equivalent to a **-verbose** option.

To get something similar to the last behaviour, you can use the `__parent_arguments__` attribute, containing a list of parent command names, whose arguments will be inherited.

true true

### Warning false true 4

If some arguments are conflicting given their names, the latter definition of the argument will be used in priority, which should with the framework structure, always be the one of the child command.

true true

### 1.8.3.2 Advanced use case

`argparse` will always add an *help* option by default to any parser that it creates. But for inheritance, this behaviour is not always wanted, since the child command will override the *help* command of the parent parser. To avoid such situations, you can also create a parser that will not be used, only as a parent for other parsers. For this, you only need to override the `parser()` of the `Command`.

```
def parser(self, subparser, parents):
    """
    Return the parser for the command and options that this command must
    use. Take as argument the subparser from the parent parser.
    """
    # create a parser with no help
    parser = argparse.ArgumentParser(add_help=False)


    # add the parser to the list of parents
    new_parents = parents + [parser]

    # call the Command class method
    old_parser = super(WhoCommand, self).parser(subparser, new_parents)

    # return this parser
    return parser
```

This way, a new parser is created with its arguments and no help, given to the parser of the command as a parent to get its options, and the parser is returned to be associated to the command, if a child wants to reference its arguments.

true true

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>33</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>33</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>33</b> |  |                    |                       |                    |                              |

## Warning falsetrue5

By using this technique, you are modifying the intrinsic behaviour of the framework. While it can be helpful some times, if you don't know what you are doing or what you want to do can be achieved in an other way, try to avoid this "advanced technique", since side effects will be unknown. ■

truetrue

## 1.8.4 API

**class** `poppy.core.command.Command`

Bases: `object`

Base class for all accepted commands for the pop command line program.

**add\_arguments** (*parser*)

Used by the user to add arguments associated to the given parser.

**classmethod** **add\_parent\_parser** (*name, parser*)

Used to add a parser with its options and be able to refer from a command, since the conflict handler of argparse is not well done, as many other things.

**has\_children** ()

To know if the command has subcommands.

**parser** (*subparser, parents*)

Return the parser for the command and options that this command must use. Take as argument the subparser from the parent parser.

**print\_help** ()

Print the command help message

**setup\_tasks** (*pipeline*)

Set up the task workflow

**subparser** (*parser*)

Should return a subparser for this command. Not always called, just when the command has possibly subcommands to generate.

**class** `poppy.core.command.CommandManager`


A class to manage the available defined commands by the user through the plugin command classes.

**add** (*name, cls*)

Adds the command and its class to the manager.

**create** (*instance*)

Register the instances of the commands by their name.

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>34</b> |

## **generate** (*options*)

Given a first base subparser and the parents parser, generate the parsers for children commands recursively.

## **launch** (*argv=None*)

Launch the commands by parsing the input of the program and then calling the good command with the good arguments.

## **parse** (*argv=None*)

Responsible to parse the command line, and also check the consistency of the tree of commands for the base command.

## **preprocess\_args** (*argv*)

Preprocess options like `--settings` and `--config`.

These options could affect the pipeline behavior, so they must be processed early.

**Parameters** *argv* –

**Returns** options, args

# 1.9 Descriptor

## 1.9.1 Introduction

A pipeline built with the POPPy framework usually needs several plugins to generate different data. For traceability, the pipeline needs to track the versions of the plugins, data format used to generate data products. This is done with the registration of the plugins information through a *descriptor* file, providing the interface and information necessary to the pipeline.

The following sections describe the *descriptor* interface and the process for loading plugins using the descriptor file.


## 1.9.2 Descriptor interface

There is two kind of descriptors:

- one for the pipeline,
- one for each plugins.

### 1.9.2.1 Pipeline descriptor

The pipeline descriptor provides metadata associated to the pipeline, databases and used plugins. All information is inside `pipeline`.

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>35</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>35</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>35</b> |  |                    |                       |                    |                              |

## 1.9.2.1.1 Identification

The identity of the pipeline is defined with the following JSON objects:

- `identifier`: the identifier of the pipeline.
- `name`: a human readable name for the pipeline.
- `description`: the purpose of the pipeline.

## 1.9.2.1.2 Release

The `release` object shall inform about the current S/W release. It shall contain the following attributes:

- `version`: current version of the pipeline in the format ‘MAJOR.MINOR.REVISION’, following the ROC conventions [AD2].
- `date`: date and hour of the release of the S/W in the format ‘YYYY-MM-DD’, where ‘YYYY’, ‘MM’ and ‘DD’ are respectively the year, month and date of the release.
- `author`: name of the person, team or entity responsible of the release.
- `contact`: contact of the author (e.g., email)
- `institute`: name of the institute that delivers the release.
- `modification`: a string containing the list of S/W modifications in the current release.


In addition, the `release` object can provide the following optional attributes:

- `file`: file name to reference a data schema, master CDF, etc.
- `reference`: name of a file as reference for the documentation, used as an indication for the ROC team.
- `url`: indication for an online resource.

## 1.9.2.1.3 Project

The `project` field contains information used for the auto-generation of some CDF skeleton files.

- `name`: name of the project as to set in the skeleton.
- `source`: the source of the data.
- `provider`: the provider of the data.
- `discipline`: category of the data.
- `PI`: informations on the Principal Investigator.
  - `PI.name` for the name of the PI.

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>36</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>36</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>36</b> |  |                    |                       |                    |                              |

– affiliation for where the PI is affected.

- `instrument_type`: the instrument used in the project.
- `mission_group`: the group of the mission.

#### 1.9.2.1.4 Databases

The `databases` object contains a list of object, representation of the databases used by the pipeline and their metadata for future references. Each database database follows this structure.

- `identifier`: the identifier of the pipeline.
- `name`: a human readable name for the pipeline.
- `description`: the purpose of the pipeline.
- `release`: a release object as defined for the `release` of the pipeline. The structure must be the same.

#### 1.9.2.1.5 Calibration softwares

`calibration_softwares` contains the list of paths to the external calibration softwares whose descriptor must be loaded into the ROC database.

#### 1.9.2.2 Plugin descriptor


The descriptor is a file in the JSON format located inside the `namespace/plugin/descriptor.json`.

A descriptor file is validated against a schema located inside the `poppy/pop/config/plugin-descriptor-schema.json`.

##### 1.9.2.2.1 Identification

Each S/W will be identified in the pipeline by the attributes provided in the `identification` JSON object:

- `project`: name of the project. It shall be “ROC-SGSE” for S/W used in the ROC-SGSE pipeline, and “RPW” otherwise.
- `name`: a cool name for the calibration software, to be human readable.
- `identifier`: a unique name used as reference by the ROC-SGSE pipeline to identify the S/W. It shall contain Latin alphabet uppercase letters only. For ease of use your name should be `namespace.plugin`. In all case , the developer team shall validate the identifier (ID) to avoid duplicated names.

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue<br/><b>02</b></div> <div>Revision<br/><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:<br/><b>37</b></div> </div> |
|--|----------------------------|--|

- `description`: short description of the software. This description will be saved in the ROC database.

### 1.9.2.2.2 Release

The `release` object shall inform about the current S/W release. It is also used to describe the output data (see section *Outputs*). It shall contain the following attributes:

- `version`: Current version of the S/W in the format ‘MAJOR.MINOR.REVISION’, following the ROC conventions [AD2].
- `date`: Date and hour of the release of the S/W in the format ‘YYYY-MM-DD’, where ‘YYYY’, ‘MM’ and ‘DD’ are respectively the year, month and date of the release.
- `author`: Name of the person, team or entity responsible of the release
- `contact`: contact of the author (e.g., email)
- `institute`: Name of the institute that delivers the release
- `modification`: a string containing the list of S/W modifications in the current release.

In addition, the `release` object can provide the following optional attributes:

- `file`: file name to reference a data schema, master CDF, as an indication for the ROC team. Only required in the outputs modes object descriptions.
- `reference`: name of a file as reference for the documentation, used as an indication for the ROC team.
- `url`: indication for an online resource.


### 1.9.2.2.3 Tasks

The `tasks` object contains the list of tasks that the plugin defines. For each task, its name, its purpose and the list of input/output datasets to be read/saved shall be supplied. It allows the pipeline to control if the expected output data files are correctly saved.

Each function listed in the `tasks` object shall contain the following attributes and JSON arrays:

- `name`: the name of the task. It will be used as internal reference by the pipeline.
- `description`: a short description of the purpose of the function.
- `inputs`: a JSON object containing the list of the input datasets required by the task.
- `outputs`: a JSON object containing the list of the outputs returned by the task.

The purpose and the content of the `inputs` and `outputs` are detailed in the two next sections.

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>38</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>38</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>38</b> |  |                    |                       |                    |                              |

## 1.9.2.2.4 Inputs

The pipeline requires information in order to identify the specific input parameters of a given task. This is the aim of the `inputs` object, which provides the list of the specific input parameters in JSON objects, with the two following mandatory attributes:

- `identifier`: The dataset ID associated to the input data file, as referenced in the ROC system.
- `version`: The version of the input data file. This allows the ROC pipeline to ensure that S/W will process the right version of the input file.

## 1.9.2.2.5 Outputs

The pipeline requires information in order to verify that the expected output data files have been correctly produced at the end of a given task execution.

Each JSON object listed in `code:outputs` shall described in details the corresponding dataset, using the following attributes/objects:

- `identifier`: The dataset ID associated to the output, as referenced in the ROC system. It shall be unique and comply the naming convention listed in [AD2].
- `name`: a more human-readable name for the dataset, not necessarily unique.
- `description`: a short description of the dataset.
- `level`: the processing level of the dataset. Allowed values are “LZ”, “L0”, “L1”, “L2”, “L2R”, “L2S”, “L3”, “L4”, “AUX”, “LLO”, “LL1”, “HK”.
- `release`: information about the release of the dataset. The structure is the same as defined in the section *Release*. In the case of a dataset using the CDF format, the “file” attribute shall provide the name of the master CDF file used to generate the output data files.

In any case, the POPPy framework will perform an automated validation of the descriptor file at each new release, in order to check that the file content is consistent with the ROC database information. In particular, it will verify that the datasets declared the `tasks` object are all defined in the descriptor and across S/W.

## 1.9.2.2.6 Example

An example file taken from the `tuto.texter` module.

```

1 {
2   "identification": {
3     "project": "TUTO",
4     "identifier": "tuto.texter",
5     "name": "Text maker",
6     "description": "Reads dictionnary and packets, reconstruct the text

```



# POPPy User Manual

Doc. ref.: ROC-TST-GSE-SUM-00035-LES

Issue  
**02**

Revision  
**00**

Date: May 13, 2020

falsefalsePage:  
**39**

```
7   },
8   "release": {
9       "version": "0.0.1",
10      "date": "2018-06-03",
11      "author": "Grégoire Duvauchelle",
12      "contact": "gregoire.duvauchelle@protonmail.com",
13      "institute": "LESIA",
14      "modification": "Starting version",
15      "reference": "reference_document.pdf"
16  },
17  "tasks": [
18      {
19          "name": "get_data",
20          "category": "Software execution",
21          "description": "",
22          "inputs": {},
23          "outputs": {
24              "packets": {
25                  "identifier": "TUTO-PKT-L0",
26                  "name": "Tutorial data packets",
27                  "description": "Contains the identifier of the word",
28                  "level": "L0",
29                  "release": {
30                      "author": "Grégoire Duvauchelle",
31                      "date": "2018-06-03",
32                      "version": "01",
33                      "contact": "gregoire.duvauchelle@protonmail.com",
34                      "institute": "LESIA",
35                      "modification": "Starting"
36                  }
37              }
38          }
39      },
40      {
41          "name": "load_dict",
42          "category": "Software execution",
43          "description": "Loads the dictionary in the database",
44          "inputs": {},
45          "outputs": {}
46      },
47      {
48          "name": "decommute",
49          "category": "Software execution",
50          "description": "Replace the indexes with the words for every_
↩packet",
51          "inputs": {
52              "packets": {
```



# POPPy User Manual

Doc. ref.: **ROC-TST-GSE-SUM-00035-LES**


Issue  
**02**

Revision  
**00**


Date: May 13, 2020


falsefalsePage:  
**40**

```
53         "identifier": "TUTO-PKT-L0",
54         "version": "01"
55     },
56 },
57 "outputs": {
58     "words": {
59         "identifier": "TUTO-WDS-L1",
60         "name": "Tutorial words file",
61         "description": "Contains the words",
62         "level": "L1",
63         "release": {
64             "author": "Grégoire Duvauchelle",
65             "date": "2018-06-03",
66             "version": "01",
67             "contact": "gregoire.duvauchelle@protonmail.com",
68             "institute": "LESIA",
69             "modification": "Starting"
70         }
71     }
72 },
73 },
74 {
75     "name": "make_text",
76     "category": "Software execution",
77     "description": "",
78     "inputs": {
79         "words": {
80             "identifier": "TUTO-WDS-L1",
81             "version": "01"
82         }
83     },
84     "outputs": {
85         "text": {
86             "identifier": "TUTO-TXT-L2",
87             "name": "Tutorial text file",
88             "description": "contains the text",
89             "level": "L2",
90             "release": {
91                 "author": "Grégoire Duvauchelle",
92                 "date": "2018-06-03",
93                 "version": "01",
94                 "contact": "gregoire.duvauchelle@protonmail.com",
95                 "institute": "LESIA",
96                 "modification": "Starting"
97             }
98         }
99     }
```

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>41</b> |

|     |   |
|-----|---|
| 100 | } |
| 101 | ] |
| 102 | } |

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>42</b> |

|  |                              |  |                    |                       |                    |                              |
|--|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>43</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>43</b> |
| Issue<br><b>02</b>   | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020   | falsefalsePage:<br><b>43</b> |  |                    |                       |                    |                              |

## Chapter 2

## Developer Guide

The goal of this chapter is to provide information on how POPPy works in the inside. It can be useful to add features to the framework.

### 2.1 Workflow

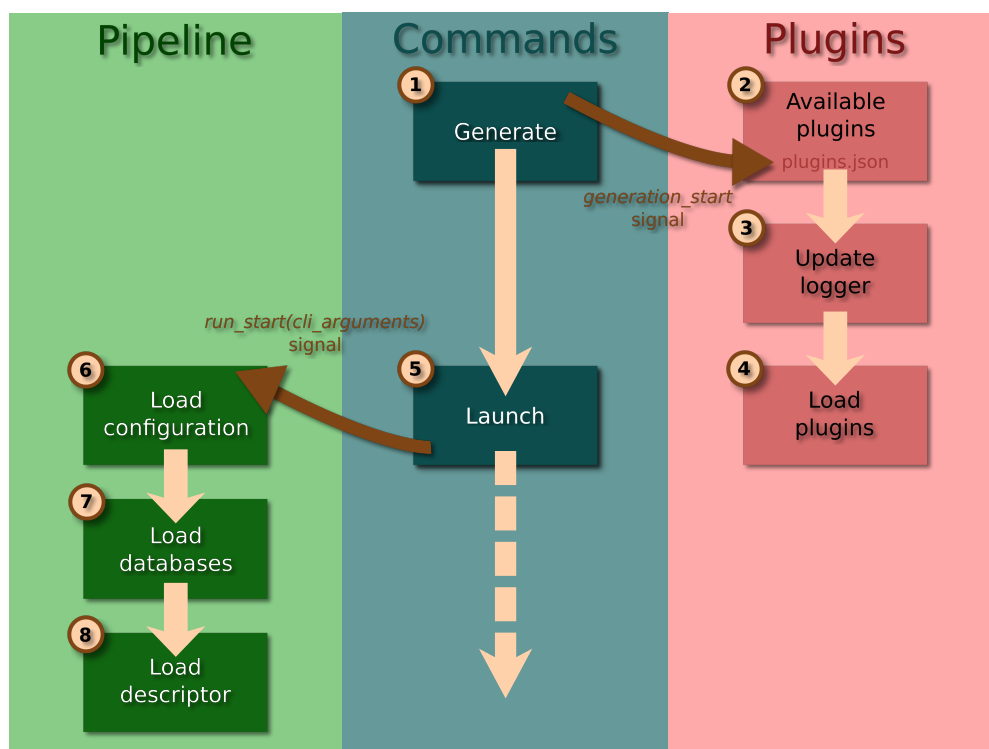



Fig. 2.1: Workflow of the launching of a POPPy pipeline

The pipeline is launched through a `poppy.pop.scripts.scripts.main()` function starting the generation of the commands with the dedicated command module and then launches the good command

|  |                            |  |
|--|----------------------------|--|
|  | <h1>POPPy User Manual</h1> | <div>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></div> <div> <div>Issue</div> <div><b>02</b></div> </div> <div> <div>Revision</div> <div><b>00</b></div> </div> <div> <div>Date: May 13, 2020</div> <div>falsefalsePage:</div> <div><b>44</b></div> </div> |
|--|----------------------------|--|

according to the content of the CLI. A representation of the process is given in Fig. 2.1.

For details on the process:

1. The `poppy.core.command.CommandManager` starts generating the commands. It sends the `poppy.core.command.CommandManager.generation_start` signal to inform connected objects that a generation started. A setup function as been connected at the import of the module to the command manager and is called before the generation.
2. In the `poppy.pop.scripts.scripts.setup()` function, the available plugins are loaded from the `settings.py` file where activated plugins are inserted. This file is in the pipeline root directory.
3. Since we introduce new code in the pipeline through plugins, we need to add our handlers of the logs into the loggers defined by the plugins (`poppy.pop.scripts.scripts.setLogger()`).
4. Plugins are loaded if we can import them correctly, and if the process of loading a plugin into the pipeline worked. See plugins.
5. All slots of the signal have been treated it, so the main function says to the `rgts_library.tools.command.CommandManager` to launch the command from the informations provided in the CLI. A signal `rgts_library.tools.command.CommandManager.run_start` with the parsed CLI arguments in argument of the signal is emitted. An `poppy.pop.scripts.scripts.pipeline_init()` function has already been connected to this signal at module importation.
6. The init of the pipeline starts by loading the configuration file selected on the CLI or at the default path and read the data inside it. It checks that the configuration file is valid against the schema.
7. From information inside the configuration file, the databases are loaded (`rgts_library.tools.database.load_databases()`).
8. Finally, the descriptor is also loaded. With it, the pipeline can access all information on versions for softwares as needed.

Then the selected command takes the control and run.

## 2.1.1 API

`poppy.pop.scripts.scripts.main(argv=None)`

The main function, called when the pop program is running.

`poppy.pop.scripts.scripts.setup(options)`

Responsible of the setup before the generation of commands.


`poppy.pop.scripts.scripts.load_config(config_path, schema_path)`

Load the data in the configuration file and set it in the class.

`poppy.pop.scripts.scripts.load_descriptor(path, schema_path)`

Load the data in the descriptor file and set it in the class.

## 2.1. WORKFLOW

|  |                            |   |                              |
|--|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>45</b> |

`poppy.pop.scripts.scripts.set_config(args)`

Set the file name where are stored the configuration parameters.

`poppy.pop.scripts.scripts.set_descriptor(args)`

Set the file name where are stored the descriptors.

`poppy.pop.scripts.scripts.pipeline_init(args)`


Used to init some things in the pipeline, without interacting with it directly, allowing to reuse the commands defined for the pipeline from an other environment where the databases, configuration, etc are already set.


`poppy.pop.scripts.scripts.pipeline_init(args)`

Used to init some things in the pipeline, without interacting with it directly, allowing to reuse the commands defined for the pipeline from an other environment where the databases, configuration, etc are already set.

`poppy.pop.scripts.scripts.setup(options)`


Responsible of the setup before the generation of commands.


|  |                   |   |                              |
|--|-------------------|---|------------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>46</b> |

|   |                   |   |                              |
|---|-------------------|---|------------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|   |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|   |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>47</b> |

## Chapter 3

### TODO list


|  |                   |   |                              |
|--|-------------------|---|------------------------------|
|  | POPPy User Manual | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|  |                   | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|  |                   | Date: May 13, 2020                          | falsefalsePage:<br><b>48</b> |

|   |                            |   |                              |
|---|----------------------------|---|------------------------------|
|  | <h1>POPPy User Manual</h1> | Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b> |                              |
|   |                            | Issue<br><b>02</b>                          | Revision<br><b>00</b>        |
|   |                            | Date: May 13, 2020                          | falsefalsePage:<br><b>49</b> |

# Python Module Index

## p

`poppy.core.command`, 33  
`poppy.core.task`, 24  
`poppy.pop.loop`, 17  
`poppy.pop.pop`, 17  
`poppy.pop.scripts.scripts`, 44

|   |                              |  |                    |                       |                    |                              |
|---|------------------------------|--|--------------------|-----------------------|--------------------|------------------------------|
|  | <h1>POPPy User Manual</h1>   | <p>Doc. ref.: <b>ROC-TST-GSE-SUM-00035-LES</b></p> <table><tr><td>Issue<br/><b>02</b></td><td>Revision<br/><b>00</b></td></tr><tr><td>Date: May 13, 2020</td><td>falsefalsePage:<br/><b>50</b></td></tr></table> | Issue<br><b>02</b> | Revision<br><b>00</b> | Date: May 13, 2020 | falsefalsePage:<br><b>50</b> |
| Issue<br><b>02</b>  | Revision<br><b>00</b>        |  |                    |                       |                    |                              |
| Date: May 13, 2020  | falsefalsePage:<br><b>50</b> |  |                    |                       |                    |                              |

# Index

## Symbols

`__weakref__` (poppy.core.task.TaskError attribute), 24

## A

`add()` (poppy.core.command.CommandManager method), 33

`add_arguments()` (poppy.core.command.CommandManager method), 33

`add_parent_parser()` (poppy.core.command.CommandManager class method), 33

## C

`Command` (class in poppy.core.command), 33

`CommandManager` (class in poppy.core.command), 33

`create()` (poppy.core.command.CommandManager method), 33

## G

`generate()` (poppy.core.command.CommandManager method), 33

## H

`has_children()` (poppy.core.command.CommandManager method), 33

## L

`launch()` (poppy.core.command.CommandManager method), 34

`load_config()` (in module poppy.pop.scripts.scripts), 44

`load_descriptor()` (in module poppy.pop.scripts.scripts), 44

## M

`main()` (in module poppy.pop.scripts.scripts), 44

## P

`parse()` (poppy.core.command.CommandManager method), 34

`parser()` (poppy.core.command.CommandManager method), 33

`pipeline_init()` (in module poppy.pop.scripts.scripts), 45

`poppy.core.command` (module), 33

`poppy.core.task` (module), 24

`poppy.pop.loop` (module), 17

`poppy.pop.pop` (module), 17

`poppy.pop.scripts.scripts` (module), 44

`preprocess_args()` (poppy.core.command.CommandManager method), 34

`print_help()` (poppy.core.command.CommandManager method), 33

## S

`set_config()` (in module poppy.pop.scripts.scripts), 45

`set_descriptor()` (in module poppy.pop.scripts.scripts), 45

`setup()` (in module poppy.pop.scripts.scripts), 44, 45

`setup_tasks()` (poppy.core.command.CommandManager method), 33

`subparser()` (poppy.core.command.CommandManager method), 33

## T

`TaskError`, 24