RPW Operation Centre

Plugin-Oriented Pipeline for Python (POPPy) Framework User Manual

ROC-TST-GSE-SUM-00035-LES
Iss.01, Rev.00

Prepared by:	**Function:**	**Signature:**	**Date**
Manuel Duarte	RPW Ground Segment Software Engineer		24/04/2016
Verified by:	**Function:**	**Signature:**	**Date**
Xavier Bonnin	RPW Ground Segment Software Manager		24/06/2016
Approved by:	**Function:**	**Signature:**	**Date**
Name	RPW Ground Segment Project Manager		Dd/mm/yyyy
For application:	**Function:**	**Signature:**	**Date**
Name	Team Member #4		Dd/mm/yyyy

CLASSIFICATION **PUBLIC** ☒ **RESTRICTED** ☐

l'Observatoire de Paris LESIA
Laboratoire d'Études Spatiales et d'Instrumentation en Astrophysique

CNRS-Observatoire de PARIS
Section de MEUDON – LESIA
5, place Jules Janssen
92195 Meudon Cedex – France

Change Record

Issue	Rev.	Date	Authors	Modifications
00	00	24/11/2015	Manuel Duarte	First draft
01	00	24/06/2016	Xavier Bonnin	First release

Acronym List

Acronym	Definition
AIT	Assembly, Integration Tests
AIV	Assembly, Integration Validations
API	Application Programming Interface
CDF	Common Data Format
CIRD	Concept and Implementation Requirements Document
CNES	Centre National d'Etudes Spatiales
CoI	Co-Investigator
CP	Cruise Phase
EDDS	EGOS Data Dissemination System
ESA	European Space Agency
ESAC	European Space Astronomy Centre
ESOC	European Space Operation Centre
GUI	Graphical User Interface
HF	High Frequency
HFR	High Frequency Receiver
IAP	
ICD	Interface Control Document
IDL	Interactive Data Language

IOR	Instrument Operation Request
IRF	Institutet för rymdfysik
JSON	JavaScript Object Notation
LESIA	Laboratoire d'Etudes Spatiales et d'Instrumentation en Astrophysique
LF	Low Frequency
LFR	Low Frequency Receiver
LL	Low Latency
LPC2E	Laboratoire de Physique et Chimie de l'Environnement et de l'Espace
LPP	Laboratoire de Physique des Plasmas
MADAWG	Modeling And Data Analysis Working Group
MOC	Mission Operation Centre
NEOP	Near Earth Operation Phase
NP	Nominal Phase
ORM	Object Relational Mapping
OS	Operating System
PI	Principal Investigator
POPPy	Plugin-Oriented Pipeline for Python
RDBMS	Relational Database Management System
RGS	RPW Ground Segment
RGTS	ROC Ground Test SGSE
ROC	RPW Operation Centre
RPL	RPW Packet parsing Library
RPW	Radio and Plasma Waves
SBM	Selected Burst Mode
SCM	Search-Coil Magnetometer
SDP	Software Development Plan
SGSE	Software Ground Support Equipment
SOC	Science Operation Centre
SolO	Solar Orbiter
SOOP	Solar Orbiter Operation Plan
SSH	Secure SHell
SSMM	State Solid Mass Memory
SUM	Software User Manual
S/C	Spacecraft
S/W	Software

TBC	To Be Confirmed
TBD	To Be Defined
TBW	To Be Written
TC	Telecommand
TDS	Time Domain Sampler
THR	Thermal Noise and High Frequency Receivers
TM	Telemetry
TNR	Thermal Noise Receiver
URD	User Requirement Document
XML	eXtended Markup Language

Table of Contents

List of figures

List of tables

Aucune entrée de table d'illustration n'a été trouvée. Dans le document, sélectionnez les mots à inclure dans la table des matières, puis, sur l'onglet Accueil, sous Styles, cliquez sur un style d'en-tête. Répétez l'opération pour chaque en-tête à inclure, puis insérez la table des matières dans le document. Pour créer manuellement une table des matières, sur l'onglet Éléments de document, sous Table des matières, pointez sur un style, puis cliquez sur la flèche vers le bas. Cliquez sur un des styles sous Table des matières manuelle, puis tapez les entrées manuellement.

1 GENERAL

1.1 Scope of the Document

This document is the software user manual (SUM) of the Plugin-Oriented Pipeline for Python framework (POPPy). POPPy has been first implemented into the ROC-SGSE software [RD1]; a tool dedicated to the analysis of the data produced during the RPW ground calibration campaigns.

Both, POPPy and ROC-SGSE are developed and maintained by the RPW Operation Centre (ROC) in charge of the RPW ground segment activities at the LESIA (Meudon).

This SUM describes the POPPy framework and how to use it with the ROC-SGSE. Especially, it addresses examples of utilization for developers and supplies information on how to add features to the ROC-SGSE using POPPy, how to apply installation procedures and how to start with the framework.

1.2 Applicable Documents

This document responds to the requirements of the documents listed in the following table:

Mark	Reference/Iss/Rev	Title of the document	Authors	Date
AD1				
AD2				

1.3 Reference Documents

This document is based on the documents listed in the following table:

Mark	Reference/Iss/Rev	Title of the document	Authors	Date
RD1	ROC-TST-GSE-SPC-00004-LES/1/1	ROC-SGSE Software Design Document	Xavier Bonnin	14/06/2016
RD2	ROC-TST-GSE-NTT-00021-LES/1/2	ROC-SGSE Test Database Description	Xavier Bonnin	10/06/2016
RD3	https://docs.python.org/3.4/	Python 3.4.5 Documentation	Python Software Fundation	25/06.2016
RD4	https://docs.python.org/3.4/install/index.html	Installing Python Modules (Legacy version)	Greg Ward	25/06/2016
RD5				
RD6				
RD7				

2 INTRODUCTION

2.1 Context and philosophy

In the framework of the RPW ground segment activities, the ROC team has to develop and run software, which are mainly able to:

- Retrieve and process RPW packet data
- Produce calibrated science data files
- Support the monitoring and analysis of the instrument data, using dedicated graphical user interfaces (GUI)

These software support capabilities must be available for both the ground calibration campaigns as well as the Solar Orbiter (SolO) mission.

In consequence the ROC team has decided to develop its software system on a common programming language, Python 3 [RD3], and common software architecture. This approach has the double advantage to optimize the design and the validation of the software and resulting data concepts, but also to minimize the time of development.

2.2 The ROC-SGSE

The POPPy framework has been first developed and used for the ROC-SGSE tool.

The ROC-SGSE is composed of three main components:

- The ROC-SGSE pipeline (also called the Ground-Test SGSE; GT-SGSE), the backend part of the software in charge of retrieving and processing the data generated during the ground calibrations
- The Test Viewer-SGSE (TV-SGSE), a GUI dedicated to the visualization of the data processed by the ROC-SGSE pipeline.
- The RPW Packet Parsing Library (RPL), the software library in charge of parsing the RPW Telemetry (TM) and Tele-command (TC) packets, using the RPW Instrument Database (IDB)

The ROC-SGSE relies on a dedicated ROC-SGSE Test Database (ROC TDB) [RD2], managed with the PostgreSQL RDBMS.

2.3 The Plugin-Oriented Pipeline for Python (POPPy) framework concept overview

The POPPy framework main concept is based on a pipeline that can be seen as a network of homogeneous modules called “plugins”. Each plugin has specific tasks to apply (e.g., retrieving data, producing output files, etc.) and is built on the same standard architecture imposed by the framework.

Note that in practice, POPPy has been designed to allow developers to deploy and generate a pipeline using the Python package mechanism [RD4].

The way the pipeline and its plugins are built, can exchange information and can be run, is presented in the next sections.

2.3.1 Main items to be found in a pipeline built with the POPPy framework

The POPPy framework relies on the following items to build and run workflows (hereafter also called "topology") through the pipeline:

- ***Descriptor*** – JSON format files to identify and to provide meta-data about both pipeline and plugins levels
- ***Tasks*** – Python classes, which allow developers to define in a standard way the tasks to be performed by a given plugin
- ***Targets*** – Python classes, which permit to define in a standard way the task inputs/outputs.

The Descriptor, Tasks and Targets items are respectively described in the sections

2.3.2 Data exchange in a pipeline build with the POPPy framework

There are three main processes to exchange data and information through a POPPy pipeline:

- Using the task *targets*.
- Using the *commands*
- Using the *signals*

These data exchange processes are detailed in the next sections.

2.3.3 POPPY mandatory plugins

2.3.3.1 Pipeline Operations Planner (POP)

The Pipeline Operations Planner (POP) is the main plugin that must be always present in a POPPy pipeline. Relying on the POPPy core library, it supports the main pipeline tasks and commands.

Especially POP supports the monitoring of the plugin activity and data processing (e.g., start/stop times, exceptions, input/output data status), which can be reported into the ROC TDB. This capability is also managed using a plugin.

2.3.3.2 Pipeline mapper (Piper)

The Pipeline mapper (Piper) is used to initialize the pipeline, by storing required description metadata into the database.

2.3.4 POPPY optional plugins

2.3.4.1 Pipeline UNit Keeper (PUNK)

The Pipeline unit keeper (PUNK) is an optional POPPy plugin, which allows users to monitor and report the pipeline activity.

2.4 ROC-SGSE use case

There are three main steps for the pipeline management:

1. Define the targets (inputs/outputs, I/O) of tasks,
2. Define tasks to perform,
3. Create dependencies between tasks and set the topology of the pipeline.

Moreover, the pipeline uses plugins to easily extend its functionalities. Thus a set of tools is available to add automation on the main steps of the pipeline creation.

The following sections describe how the framework of the pipeline works to do each step. But let start by an example using plugins to hide some details for now.

2.4.1 Typical ROC-SGSE plugin source code applying the POPPy framework convention

```
from rgts.pop import Pop from rgts.pop.plugins import Plugin

# args is supposed to contain the arguments to give to the pipeline. This
# is just a variable whose attributes are the parameters

 args = get_args_from_somewhere()

# create an instance of the pipeline

pipeline = Pop(args)

# create a task class from the information provided by a plugin. See the

# dedicated section for more details on the interface. This is not an

# instance of the task, it will give the possibility to create tasks

# following a model

Task = Plugin.manager["MyPlugin"].task("save_princess_peach")

# now define two tasks from function

@Task.as_task

def jump(task):

      """

      Take the instance of the task that will be created by the pipeline as
an argument.

       """

      # the task instance contains the pipeline instance to access some

      # properties through it and also interact with it

      parameter = task.pipeline.properties.parameter

      # do some job for jump

      return

@Task.as_task(outputs=["the_output"]) def say(task):

      """ This task will write something in the output, accessible through
the 'the_output' attribute as a property of the pipeline. """ # get the
target class, from what has been defined in the plugin

      Target = task.target("the_target")

      # create an instance of the target

      target = Target("path_for_output.ext")

       # say that we are using the target, allowing handling errors, etc

      with target.activate():
```

```
            # do things with the file

# instantiate tasks, create the topology of the pipeline and associate to
# the pipeline instance

start = jump()

pipeline | start | say()

# define which task is the starting point of the pipeline. If you create #
more complex topologies in advance, it allows to reduce the job performed
# by the pipeline # this way, jump task will be called first, and if say's
inputs exists, say # task will then be executed to (and if jump do not
have an error)

pipeline.start = start

# now you can run the pipeline to execute tasks

pipeline.run()
```

This seems a little mystical at this time, but it will be explained in the following sections. Interactions with the database and error handlings are hidden to the user, permitting to focus on the job performed by the task. All is done through the POPPY framework of the ROC-SGSE pipeline, which is provided by the `rgts.pop` module.

2.4.2 ROC-SGSE pipeline primary command

To launch the pipeline:

```
$ pop
```

If `pop` is the command provided by the pipeline installation. 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 is displayed when launching the pipeline, some jobs have al- ready done by `rgts.pop`. The workflow of the launching is described in the following section.

2.4.3 ROC-SGSE pipeline initialization workflow using POPPy

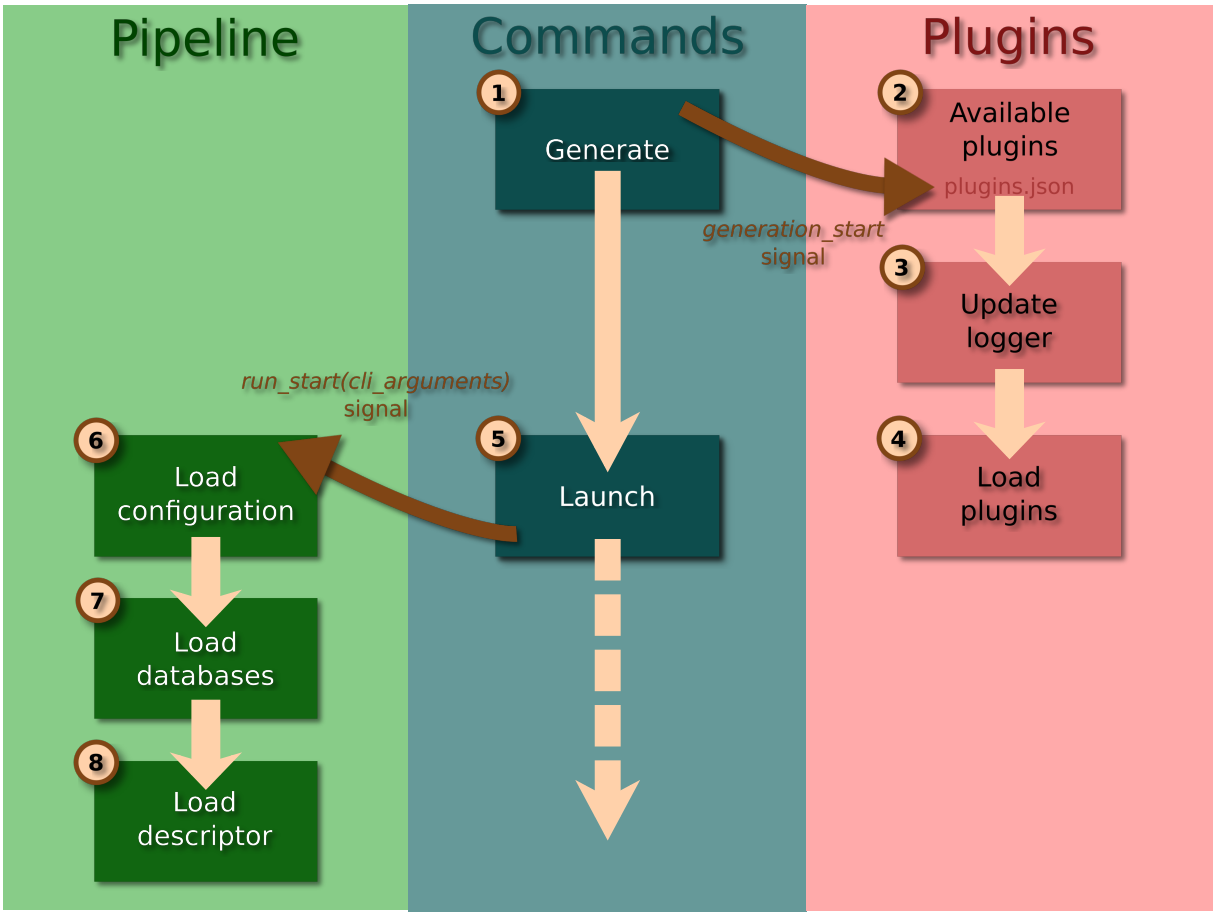

Figure 1. Workflow of the launching of the ROC-SGSE pipeline.

The ROC-SGSE pipeline is launched through a `rgts.pop.scripts.scripts.main()` function starting the generation of the commands with the dedicated command module and then launches the good command according to the content of the CLI. A representation of the process is given in the Figure 1.

For details on the process:

1. The `rgts_library.tools.command.CommandManager` starts generating the commands. It sends the `rgts_library.tools.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 `rgts.pop.scripts.scripts.setup()` function, the available plugins are loaded from the `plugins.json` where activated plugins are inserted. This file is under `rgts/pop/config/` 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 (`rgts.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 information 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 *rgts.pop.scripts.scripts.pipeline_init()* function has already been con- nected 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 software as needed.

Then the selected command takes the control and run.

2.4.4 ROC-SGSE Application Programming Interface (API)

`rgts.pop.scripts.scripts.main()` The main function, called when the pop program is running.

`rgts.pop.scripts.scripts.loadPlugins()` Read the file of plugins at the fixed place and then load plugins with the specific work to do for the models loading, commands, etc.

`rgts.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.

`rgts.pop.scripts.scripts.setup()` Responsible of the setup before the generation of commands.

`rgts.pop.scripts.scripts.setLogger(plugins)` Create a logger with the correct name for the module in order to see what happens in the program in different handlers (console, console in GUI, activity file, etc.).

3 DESCRIPTOR FILE

3.1 Introduction

A POPPy pipeline uses plugins for creating the data products. For traceability reason, the pipeline needs to track the versions of the plugins, and the data format used to generate the data products. This is done with the registration of the plugins information through a dedicated *descriptor* file in the JSON format. This file provides the interface and information required to start and run the pipeline.

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

3.2 Descriptor file interface

There are two kinds of descriptor files:

- A "*roc_pip_descriptor.json"* descriptor file for the pipeline it-self
- A "*roc_sw_descriptor.json"* descriptor file for each of the pipeline plugin and external software unit

3.2.1 Pipeline descriptor file

The pipeline descriptor file provides metadata associated to the pipeline, databases and used plugins/software. All information is inside a main JSON object called `pipeline`.

3.2.1.1 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`: a short description of the pipeline.

3.2.1.2 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.

3.2.1.3 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`:
  - `name`: the name of the PI.

- `affiliation`: the PI affiliation.

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

3.2.1.4 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 database.
- `name`: a human readable name for the database.
- `description`: A short description of the database.
- `release`: A release object as defined for the `release` of the pipeline. The structure must be the same.

3.2.1.5 Modules

The `modules` object contains the list of plugin, referenced by their identifier, whose descriptor must be loaded into the ROC database.

3.2.1.6 Calibration software

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

3.2.2 Plugin descriptor file

The descriptor is a file in the JSON format located inside the `config/roc_sw_descriptor.json`, where `config/` is at the root of the directory of the plugin.

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

3.2.2.1 Identification

Each plugin 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 “ROC” otherwise.
- `name`: a human readable name for plugin.
- `identifier`: a unique name used as reference by the ROC-SGSE pipeline to identify the plugin. It shall contain Latin alphabet uppercase letters only. The hyphen character shall be used as separator if required. In all case, the ROC team shall validate the identifier (ID) to avoid duplicated names.
- `description`: A short description of the plugin. This description will be saved into the ROC database by the piper plugin.

3.2.2.2 Release

The `release` object shall inform about the current S/W release. It is also used to describe the output data (see the section 3.2.2.5). It shall contain the following attributes:

- `version`: Current version of the S/W in the format 'MAJOR.MINOR.REVISION', following the ROC conventions [RD5].
- `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.

3.2.2.3 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.

3.2.2.4 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.

3.2.2.5 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.
- :code:`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", "LL0", "LL1", "HK".
- `release`: information about the release of the dataset. The structure is the same as defined in the section 3.2.2.2. 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 ROC pipeline 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.

3.2.3 Example of descriptor file

An example file taken from the rgts.dare module.

```
{
    "identification": {
        "project": "ROC-SGSE",
        "name": "Data Requester SGSE",
        "identifier": "DARE-SGSE",
        "description": "Module to request and download new incoming test
data from MEB GSE database"
    },
    "release": {
        "version": "0.4.0",
        "date": "2016-11-03",
        "author": "Xavier Bonnin",
        "contact": "xavier.bonnin@obspm.fr",
        "institute": "LESIA",
        "modification": "Add --clear-files option to clear commands",
        "reference": "ROC-TST-GSE-SPC-00004-LES"
    },
```

```
    "tasks": [
        {
            "name": "extract_test",
            "category": "Software execution",
            "description": "Extract tests from the ROC database and store
them internally",
            "inputs": {},
            "outputs": {}
        },
        {
            "name": "parse_xml",
            "category": "Software execution",
            "description": "Parse a xml test log file",
            "inputs": {
                "xml_test_log": {
                    "identifier": "ROC-SGSE_LZ_MEB-SGSE-TEST-LOG",
                    "version": "01"
                }
            },
            "outputs": {}
        },
        {
            "name": "clear_test",
            "category": "Software execution",
            "description": "Clear a test in the ROC database",
            "inputs": {},
            "outputs": {}
        },
        {
            "name": "to_xml_file",
            "category": "Software execution",
            "description": "Get data from the MEB-GSE database and put it
into an XML file test_log",
            "inputs": {
            },
            "outputs": {
```

```
                "xml_test_log": {
                    "identifier": "ROC-SGSE_LZ_MEB-SGSE-TEST-LOG",
                    "name": "ROC-SGSE_LZ_MEB-SGSE-TEST-LOG",
                    "description": "Test log dataset",
                    "level": "LZ",
                    "release": {
                        "author": "Manuel Duarte",
                        "date": "2015-11-18",
                        "version": "01",
                        "contact": "manuel.duarte@obspm.fr",
                        "institute": "LESIA",
                        "modification": "Starting"
                    }
                }
            }
        },
        {
            "name": "copy_lz",
            "category": "Software execution",
            "description": "Copy Lz file into the outputdir with from_xml
command",
            "inputs": {
                "xml_test_log": {
                    "identifier": "ROC-SGSE_LZ_MEB-SGSE-TEST-LOG",
                    "version": "01"
                }
            },
            "outputs": {}
        }
    ]
}
```

4 PIPELINE

4.1 Usage

An example on how to use the pipeline is provided in pipeline_usage and will be used as reference. The steps are the following (see figure 2):

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.

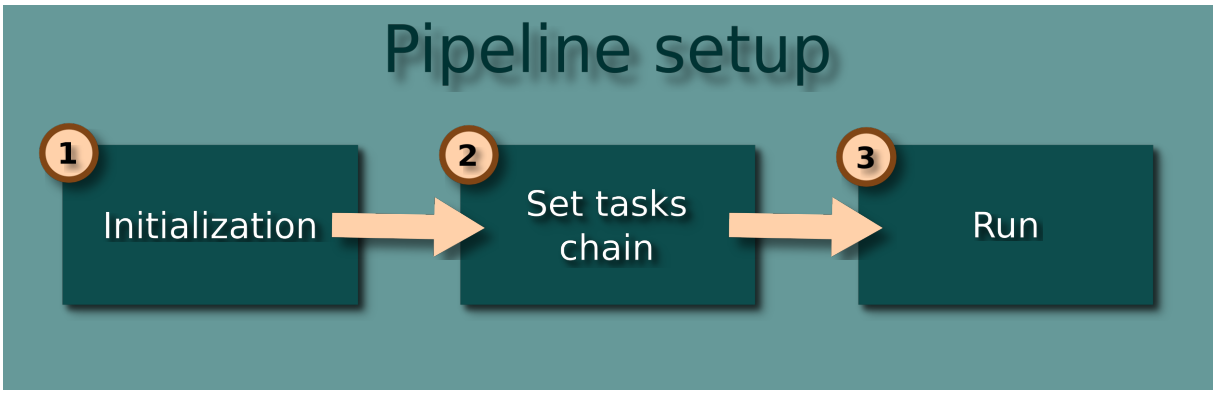

Figure 2. Steps of the pipeline setup.

A detailed description of each step is done in the following sections

4.2 Initialization

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

4.2.1 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 rgts.pop.pop.Pop.properties, and is an instance of rgts_library.tools.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:

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

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

4.2.2 Connector setup

From the informations provided by the arguments now stored into the context, the pipeline can setup the connectors. They are defined in the `connections` field of the configuration. 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 rgts_library.tools.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.

A connection as the following format in the configuration file:

```
{
    "name": "ROC",
    "database": "ROC-TDB",
    "connector": "rgts.pop.roc_connector.ROC"
}
```

- `name` is the identifier that is used as reference in the code to get the associated connector.
- `database` 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.
- `connector` is optional. If present, the class in the provided module path will be used to construct the connector, else the default rgts_library.tools.connector.Connector is used. It can be useful to add other behaviour to a given connector.

4.2.3 Dry run setup

The dry run object rgts_library.tools.dry_runner.DryRunner, a singleton in the program, is referenced by the pipeline to enable/disbale 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 ROC pipeline intact.

The state of the dry run mode is setup according to the status of the --dry-run, from the CLI of the `pop` command.

```
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.

```python
@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()
```

4.3 Task chain

4.3.1 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.

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 .rgts.pop.pop.Pop.start that gives really an entry point.

4.3.2 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 rgts.pop.pop.Pop.start and rgts.pop.pop.Pop.end attributes are existing. The rgts.pop.pop.Pop.start attribute must be set to indicate the starting point for the chain of tasks to execute. The rgts.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.

4.3.3 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 rgts.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 rgts.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 rgts.pop.task.Task.errored signal, emitted each time that an error occurred in a task inside the loop chain, to handle correctly an error while in the loop.

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

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 pipeline_loop gives an idea on what have to be done on each signal emitted by tasks.

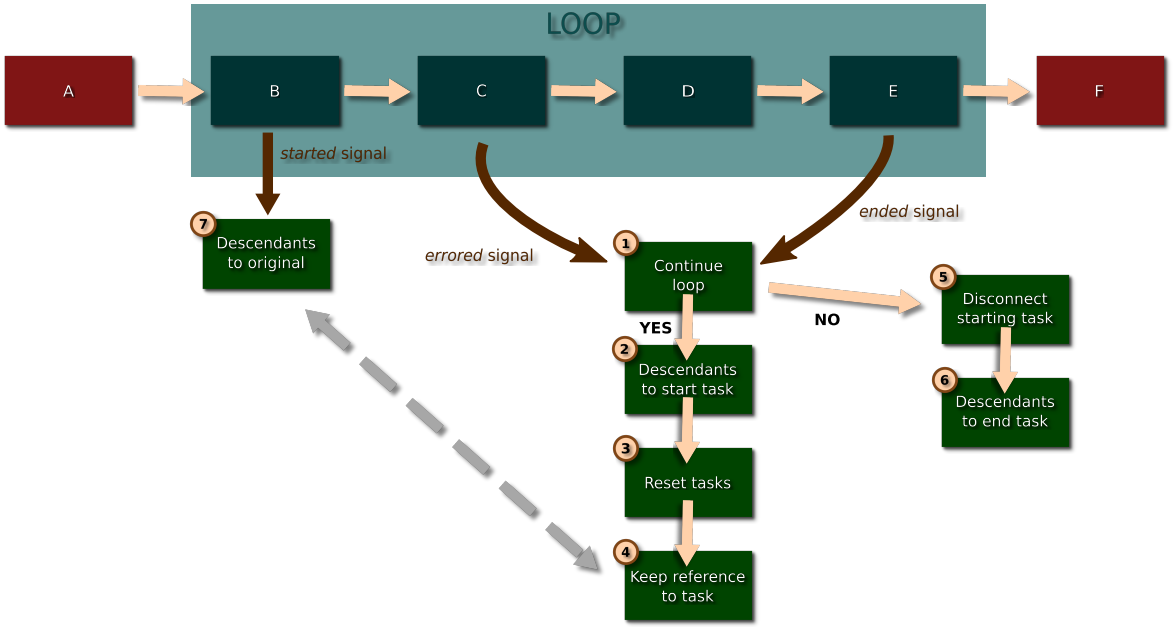

Figure 3. Flowchart for the loop of the pipeline for a chain of 6 tasks. Tasks B, C, D, E are placed inside a loop. The loop instance changes the descendants of tasks dynamically in function of the status of tasks inside the loop.

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 `errored` 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.
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.

4.4 Run

4.4.1 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.

4.4.2 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 th database for creating the mapping. This is done at the rgts_library.tools.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.

4.4.3 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 as 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.

4.5 API

class rgts.pop.pop.**Pop**(*args*) Bases: object

Pop is the class to manage all the task in the pipeline. The module pop is responsible for a few more jobs to done, such as scripts to load the pipeline and the various commands allowed to test and launch the pipeline.

All created tasks inside commands will be managed by the Pop class.

__init__(*args*) Initialization of some parameters used by the pipeline.

__or__(*task*) Used to link the first task to the pipeline and be able to loop through the task graph to run appropriately the tasks with their dependencies.

__weakref__

list of weak references to the object (if defined)

_check_existing(*targets*) Check that all the outputs of the given task exists to say that the task is completed.

_data_map(*task*, *targets*, *role*) A function to do recurrent tasks for setting the data map between jobs and I/O.

_handle_exception(*task*, *message=None*) To create an exception object for the database and store it.

_input_data_map(*task*) To create the data map informations for the inputs of a task.

_link_databases() Link databases defined in the configuration file to the connectors to made them accessible from the registry.

_output_data_map(*task*) To create the data map informations for the outputs of a task.

_run_task(*task*) To run a task and its dependencies before. As been transformed to use a stack of tasks instead of recursion, since with the loop in tasks taht it was introduced to dynamically change the topology of the pipeline, the code ended to a large number of recursive calls.

_starter(*task*) Return a context manager for automatic stuff when launching a task such setting parameters inside the database, indicating the task status into ROC database, putting I/O files, etc.

_validate_dir(*dirname*) Check if the directory is existing.

create_exception(*task*, *message*) To create the representation of an exception in the database and store it.

dry_run

To change the state of the dry run mode for the pipeline.

If True, the dry run mode is activated, else it is deactivated. Getter the state of the dry run mode in the pipeline (True/False). Setter the state of the dry run mode in the pipeline (True/False). Type bool

end

The ending task, if different from the end of the task chain.

If the ending task is changing and a task has already been set as the ending one, the old task descendants are restored to the old state.

The descendants of the given ending task are kept to be restored later if necessary. Then the generator of the task descendants is replaced by a null generator and the new ending task is set to the given one.

Getter the selected task for the end. Setter the task in the chain that will mark the end of the pipeline. Type *rgts.pop.task.Task*

generate_topology(*task*) Generate the dependency graph of the pipeline from the entry point of the task.

loop(*start*, *end*, *generator*) Create a loop between two task in the pipeline topology, according to the values given by the provided generator.

Starts by checking if the topology of the pipeline is existing. Then regenerate the topology by security. An instance of the loop is created that will override the descendants and ancestors of the start and end tasks provided in argument, at the appropriate times.

run() Run the tasks registered into the pipeline.

run_check(*task*) Responsible to launch the task and check that inputs and outputs are correctly created.

start

The starting task, if different from the start of the task chain.

If the starting task is changing and a task has already been set as the starting one, the old task ancestors are restored to the old state.

The ancestors of the given starting task are kept to be restored later if necessary. Then the generator of the task ancestors is replaced by a null generator and the new starting task is set to the given one.

Getter the selected task for the start. Setter the task in the chain that will mark the start of the pipeline.

Type *rgts.pop.task.Task* class rgts.pop.loop.**Loop**(*pipeline*, *start*, *end*, *iterable*)

Bases: object

A class to create a loop between two tasks in the pipeline easily.

__init__(*pipeline*, *start*, *end*, *iterable*) Construct the class with informations on the looping tasks in the pipeline, and on what to iterate.

__weakref__

list of weak references to the object (if defined)

_change_next_to_start(*task*) To change the next task returned by the task.

_start_generator() Generator to return the start task every time it is called.

_to_next(*task*) To handle what to do when we need to go to a next iteration. The next iteration is the whole process of going from the starting task and the ending task.

delete() Remove the reference to the loop.

init() Init some states for the loop for all tasks belonging to it.

next() To pass to the next iteration of the loop.

reset() Make a reset of the tasks inside the loop.

starting(*start*) To make a first iteration on the iterable before starting the start task.

5 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.

5.1 Task creation

5.1.1 Using class

To create a task, you can simply derive from the base class rgts.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 rgts.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 ROC pipeline, 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.

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.

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.

5.1.2 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 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()
```

5.1.3 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 rgts.pop.plugins import Plugin

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

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

Tasks through plugins contain 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.

5.2 Communication

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

5.2.1 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")
```

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

To express the following dependency:

```
A --->
      \
       \
        C ----> D ----> E
       /
      /
B --->
```

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
```

5.2.2 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...

    print("I'm working!")

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

5.2.3 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:

```
>>> task.errored.disconnect(call_on_error)
```

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!

5.3 API

class rgts.pop.task.**Task**(software=None, category=None, description=None,

name=None) Base class for tasks to run through the pop pipeline.

Bases: object **__init__**(software=None, category=None, description=None, name=None)

Initialization of some parameters.

__or__(child) To set the task in the other side of the pipe as a child task.

__repr__() Represent the task with a name. Use the class name, except if a name is provided at instanti- ation.

__weakref__

list of weak references to the object (if defined)

ancestors() Equivalent to the descendants methods, but for the parents of the task. Maybe an utility will be find some day with it...

classmethod **as_task**(func=None, inputs=None, outputs=None) To transform a simple function or method into a Task, that can be used in the pipeline.

descendants() Give the children-descendants of a task. Those tasks are dependant of this task to be run. By default, it returns the list of children as an iterable, but it can be transformed into a generator, to be able to create loops in the pipeline, returning at a given pint of the pipeline according to some specific conditions.

error() Set the status to error.

exception(message) Create an exception in the pipeline, i.e. in the ROC database with the given message as description.

input() Return a list of the input targets necessary to this task. The list contains the names of the attributes used for storing targets into the properties object of the pipeline. The task should have a reference on it.

job() Create the job representation of the task into the ROC database.

job_changed() What to do when the status of the job has changed.

ok() Set to ok the status

output() Should return a structure containing the targets outputs of these task. If a task use them as input, it should know the structure to access to good properties.

pending() Set the status to pending.

progress() Set the status to in progress.

report(value) To change the status report of the target.

reset() To reset some attributes of the task, in order to have one new task from an existing one.

run() The method that the other tasks must override to perform their computations.

start() Indicate at which time the job started.

status(value) To change the status of the task with the given provided argument.

stop() Indicate at which time the job ended.

terminated() Set the status to terminated.

warning() Set the status to warning.

6 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.

6.1 Target creation

6.1.1 Using class

A target instance can be created simply by instantiating a rgts.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:

```
from rgts.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 section 6.2 for details.

6.1.2 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_descriptor.

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 rgts.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")
```

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

6.2 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 ROC pipeline.

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 rgts.pop.target.Target.error and reported into the database.

At startup, it is marked as rgts.pop.target.Target.ok and rgts.pop.target.Target.pending.

If while treating the file, the target is detected as empty, the rgts.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 target_api.

6.3 API

class rgts.pop.target.**Target**(*filename*, *dataset*, *version*) Bases: object

Base class for th targets of the tasks, which are the input/output used and generated by the task. A wrapper is used to easily define the I/O of each task while keeping the job of managing it easy, by hiding the database representation, the existence or not of the I/O etc.

exception **TargetEmpty** Bases: Exception

Exception to indicate that the target is empty.

__weakref__

list of weak references to the object (if defined)

Target.**__init__**(*filename*, *dataset*, *version*) Store a session for the ROC database.

Target.**__weakref__** list of weak references to the object (if defined)

Target.**activate**() To use the target as a wrapper around other files type that cannot be opened as usual.

Target.**creation_date**() To get the creation date of a file. ctime does not give the creation time but the change time. Use mtime for last modification date instead.

Target.**empty**() Set the status to empty.

Target.**error**() Set the status to error.

Target.**exists**() Check that the file exists on the system.

Target.**ok**() Set to ok the status

Target.**open**(**args*, ***kwargs*) A wrapper around the open function of the file.

Target.**pending**() Set the status to pending.

Target.**progress**() Set the status to in progress.

Target.**report**(*value*) To change the status report of the target.

Target.**size**() Return the size in bytes of the file, 0 if the file doesn't exist or there is any kind of problem.

Target.**status**(*value*) To change the status of the target with the given provided argument.

Target.**target_changed**() Recreate the representation of the target if it changed.

Target.**terminated**() Set the status to terminated.

Target.**warning**() Set the status to warning.

7 PIPELINE MANAGEMENT

Description of some frameworks for tools to manage the pipeline.

7.1 Command

7.1.1 Creating a new command

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

7.1.1.1 Simple command

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

```
from rgts.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 allow 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.

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.**

7.1.1.2 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(
            "--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_aguments() 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",
    )

    # 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.

7.1.2 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:")

        # 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.

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
```

7.1.3 Inheritance of parameters

7.1.3.1 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
```

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.

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.

7.1.3.2 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.

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.

7.1.4 API

class rgts_library.tools.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 argu- ment the subparser from the parent parser.

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

class rgts_library.tools.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.

generate() Given a first base subparser and the parents parser, generate the parsers for children com- mands recursively.

launch() Launch the commands by parsing the input of the program and then calling the good com- mand with the good arguments.

parse() Responsible to parse the command line, and also check the consistency of the tree of com- mands for the base command.

8 MIGRATION

8.1 Introduction

The pipeline provides a migration procedure to be able to easily upgrade a production pipeline database across the development.

8.1.1 Principle

Each plugin has the possibility to define a migrations directory inside the models directory. It contains python files that will be loaded by the migrator, the python class that manages the migration of all registered plugins (see rgts.pop.migration).

A migration file must define constants `REVISION` and `DOWN_REVISION`.

`REVISION` is the identifier of the revision of the database that will be used as reference by other migration files and by the migrator.

`DOWN_REVISION` refers to the parent revision. The modification to upgrade the database are applied upon the state of the `DOWN_REVISION`. For a downgrade, this is the revision that will

become current after the downgrade. Setting `None` indicates that `REVISION` is the root of the revisions (the first one).

The migration must also define two functions for the upgrade and the downgrade of the database. These functions takes in argument the task of the migrator command to be able to access some attributes of the pipeline from it if necessary.

The migrator checks that the interface of migration files is correct before launching any commands on the database.

8.1.2 Example of a migration file

```
# rgts/pop/models/migrations/004.py
REVISION = "004"
DOWN_REVISION = "003"


def upgrade(task):
    # get the database objects for the ROC
    database = task.pipeline.roc.get_database()
    database.create_engine()

    # try closing the session to kill all current transactions
    try:
        database.scoped_session.close_all()
    except Exception as e:
        logger.error("Can't close sessions on sqlalchemy")
        raise e

    # execute the script
    database.engine.execute(
        """
        ALTER TABLE data.test_log ADD COLUMN test_sha1 varchar(40) DEFAULT
'';
        COMMIT;
        """
    )


def downgrade(task):
    # get the database objects for the ROC
    database = task.pipeline.roc.get_database()
    database.create_engine()

    # try closing the session to kill all current transactions
    try:
        database.scoped_session.close_all()
    except Exception as e:
        logger.error("Can't close sessions on sqlalchemy")
        logger.error(e)
        return

    # execute the script
    database.engine.execute(
```

```
        """
        ALTER TABLE data.test_log DROP COLUMN test_sha1;
        COMMIT;
        """
    )
```

8.2 Migrator

8.2.1 Usage

Before any use of the migrator, it is necessary to build it:

```
# create the migration object
migrator = Migrator()

# build the graph
migrator.build()
```

This process goes through all plugins and get the ones that define revisions for the database at the initialization. It checks also the validity of the order of revisions while constructing the graph structure of revisions for each plugin in the rgts.pop.migration.Migrator.build method.

8.2.1.1 Get path for upgrading or downgrading

The migrator can gives all revisions between two given revisions for a plugin. This path contains rgts.pop.migration.Revision instances that can be used to manipulate the migration file and get informations on the revision.

```
# to get the path to upgrade the plugin from old_revision to the
# new_revision
path = migrator.upgrade_path(plugin, old_revision, new_revision)

There is the equivalent for the downgrade.
```

8.2.1.2 Current revision

The migrator allows to get the current revision of a plugin `rgts.pop.migration.Migrator.get_current_revision` and to set the current revision with `rgts.pop.migration.Migrator.set_current_revision`.

This information is set into the `migrator.revisions` table, with a field `revisions_software` for the name of the plugin and `revisions_revision` for the current revision of the plugin.

For each command of getting or setting the current revision, the structure of the database is automatically created into the ROC database through the *Migrator* connector.

8.2.2 Commands

The first to do with the command for the migrator is to set the current revision of a plugin if not already done. To set for a registered plugin (`POP-SGSE`):

```
$ pop roc migrate current set POP-SGSE 001
```

sets the plugin `POP-SGSE` to the revision 001 defined in the migration file of the plugin. If errors are present in the CLI, the command stops with an error.

To get the status of the revision for each software registered in the migrator:

```
$ pop roc migrate current
```

shows the current revision of plugins.

For upgrading to a new revision from the current one:

```
$ pop roc migrate upgrade POP-SGSE 004
```

the downgrade process is identical and will downgrade from the current revision.

```
$ pop roc migrate downgrade POP-SGSE
```

The command above will reset the database for the `POP-SGSE` plugin to the empty state (the none revision). If the database contains data, it will be cleared (the part of the plugin)

The downgrade command should not be used in the production database, except if a particular reason impose that. Since it can potentially erase data, it must be used with precaution.

8.3 API

class rgts.pop.migration.**Migrator** Bases: object

Handle the migration procedures for all softwares (plugins) of the pipeline.

__init__() The migrator is initialized with the plugins set by the user to store the list of revisions of each plugin.

__weakref__

list of weak references to the object (if defined)

_base_path(*plugin*, *start*, *end*) Get the list of revisions to loop on, in order to apply the upgrade for the plugin given in argument.

_get_revisions() Get plugins from the plugin manager and store a manager of revisions for each of them.

build() Build the tree of revisions for each plugin and check the consistency of the tree.

create_migrator_structure(*database*) Create the structure of the migrator if not already created.

delete_migrator_structure(*database*) Delete the structure of the migrator if already created.

get_current_revision(*plugin*) Get the current revision of a plugin.

get_status() Returns the status of the migrator database, with the plugin and their revisions.

is_structure_existing(*database*) Indicate by True or False if the migrator database structure is already present or not.

set_current_revision(*plugin*, *revision*) Set a plugin to a given revision as the current one.

valid_revision(*plugin*, *revision*) Check if a revision is a valid one, registered in the revision manager.

exception rgts.pop.migration.**MigrateError** Bases: Exception

Errors linked to the migration.

__weakref__

list of weak references to the object (if defined)

9 LIST OF TBC/TBD/TBWS

TBC/TBD/TBW			
Reference/Page/Location	Description	Type	Status

10 DISTRIBUTION LIST

LISTS See Contents lists in "Baghera Web": Project's informations / Project's actors / RPW_actors.xls and tab with the name of the list or NAMES below		Tech_LESIA
		Tech_MEB
		Tech_RPW
		[Lead-]Cols
		Science-Cols

INTERNAL

LESIA CNRS	x	M. MAKSIMOVIC
	x	Y. DE CONCHY
	x	X. BONNIN
	x	QN NGUYEN
	x	**E. HOLLE**
	x	B. CECCONI
	x	A. VECCHIO

LESIA CNRS		

EXTERNAL (To modify if necessary)

CNES		C. FIACHETTI
		C. LAFFAYE
		R.LLORCA-CEJUDO
		E.LOURME
		M-O. MARCHE
		E.GUILHEM
		J.PANH
		B.PONTET
IRFU		L. BYLANDER
		C.CULLY
		A.ERIKSSON
		SE.JANSSON
	x	A.VAIVADS
	x	Y. KHOTYAINTSEV
LPC2E		P. FERGEAU
	x	G. JANNET
		T.DUDOK de WIT
	x	M. KRETZSCHMAR
		V. KRASNOSSELSKIKH
SSL		S.BALE

AsI/CSRC		J.BRINEK
		P.HELLINGER
		D.HERCIK
		P.TRAVNICEK
IAP		J.BASE
		J. CHUM
		I. KOLMASOVA
		O.SANTOLIK
	x	J. SOUCEK
	x	L.UHLIR
IWF		G.LAKY
		T.OSWALD
		H. OTTACHER
		H. RUCKER
		M.SAMPL
		M. STELLER
LPP	x	T.CHUST
		A. JEANDET
		P.LEROY
		M.MORLOT
	x	B. KATRA