wltp Documentation

Release 0.0.9-alpha.3

Kostis Anagnostopoulos

March 01, 2016

Contents

Release 0.0.9-alpha.3

Documentation https://wltp.readthedocs.org/

Source https://github.com/ankostis/wltp

PyPI repo https://pypi.python.org/pypi/wltp

Keywords UNECE, automotive, car, cars, driving, engine, fuel-consumption, gears, gearshifs, rpm, simulation, simulator, standard, vehicle, vehicles, wltc

The *wltp* is a python package that calculates the *gear-shifts* of Light-duty vehicles running the *WLTP* driving-cycles, according to *UNECE*'s GTR (Global Technical Regulation) draft.

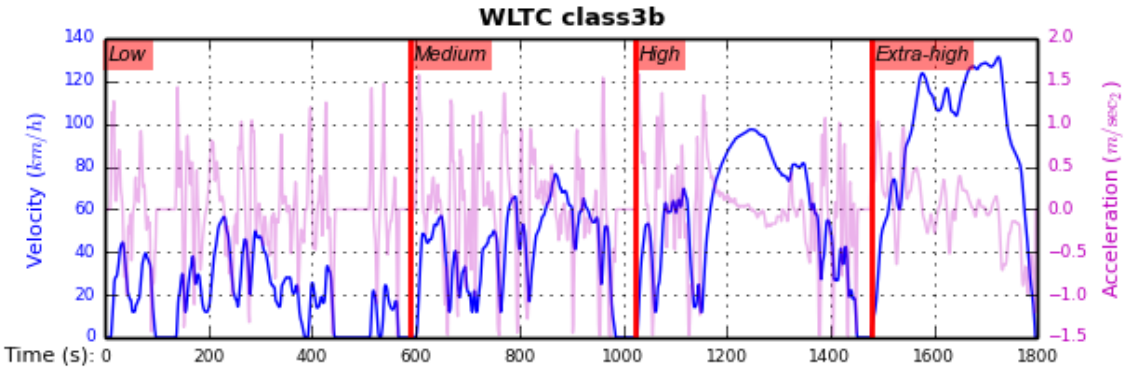


Fig. 1: **Figure 1:** *WLTP cycle for class-3b Vehicles*

Attention: This project is still in *alpha* stage. Its results are not considered "correct", and official approval procedures should not rely on them. Some of the known deficiencies are described in these places:

- In the Changes.
- Presented in the diagrams of the *Tests, Metrics & Reports* section.
- Imprinted in the `wltp_db_tests` test-case (automatically comparared with a pre-determined set of vehicles from Heinz-db on each build) Currently, mean rpm differ from Heinz-db < 0.5% and gears diff < 5% for a 1800-step class-3 cycle.

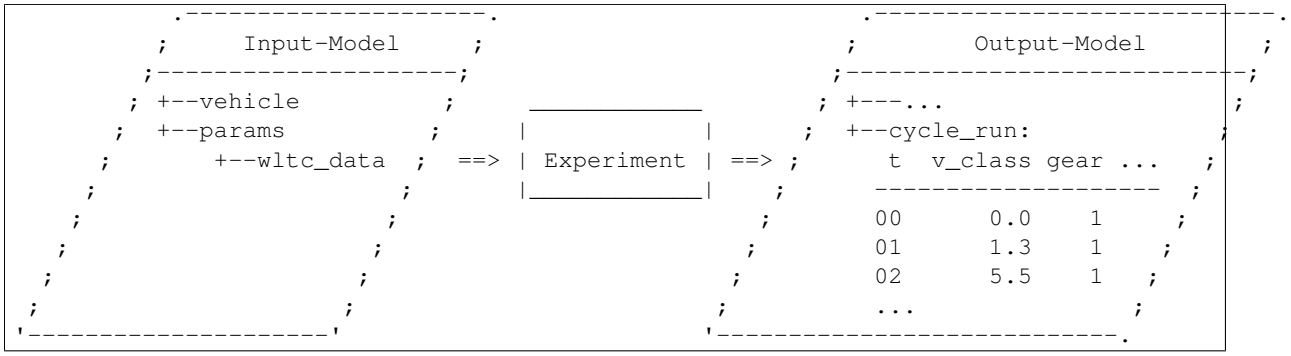
CHAPTER 1

Introduction

1.1 Overview

The calculator accepts as input the vehicle's technical data, along with parameters for modifying the execution of the *WLTC* cycle, and it then spits-out the gear-shifts of the vehicle, the attained speed-profile, and any warnings. It does not calculate any CO_2 emissions.

An "execution" or a "run" of an experiment is depicted in the following diagram:



The *Input & Output Data* are instances of *pandas-model*, trees of strings and numbers, assembled with:

- sequences,
- dictionaries,
- `pandas.DataFrame`,
- `pandas.Series`, and
- URI-references to other model-trees.

1.2 Quick-start

On *Windows*/*OS X*, it is recommended to use one of the following "scientific" python-distributions, as they already include the native libraries and can install without administrative priviledges:

- WinPython (*Windows* only),
- Anaconda,
- Canopy,

Assuming you have a working python-environment, open a *command-shell*, (in *Windows* use **`cmd.exe`** BUT ensure **`python.exe`** is in its `PATH`), you can try the following commands:

Install

```
$ pip install wltp --pre
$ wltp --winmenus                          ## Adds StartMenu-items, Windows only.
```

See: Install

Cmd-line

```
$ wltp --version
0.0.9-alpha.3

$ wltp --help
...
```

See: *Cmd-line usage*

GUI

```
$ wltp --gui`                              ## For exploring model, but not ready yet.
```

Excel

```
$ wltp --excelrun                          ## Windows & OS X only
```

See: *Excel usage*

Python-code

```
from wltp.experiment import Experiment

input_model = { ... }            ## See also "Python Usage" for model contents.
exp = Experiment(input_model)
output_model = exp.run()
print('Results: \n%s' % output_model['cycle_run'])
```

See: *Python usage*

Tip: The commands beginning with `$`, above, imply a *Unix* like operating system with a *POSIX* shell (*Linux*, *OS X*). Although the commands are simple and easy to translate in its *Windows* counterparts, it would be worthwile to install Cygwin to get the same environment on *Windows*. If you choose to do that, include also the following packages in the *Cygwin*'s installation wizard:

```
* git, git-completion
* make, zip, unzip, bzip2
* openssh, curl, wget
```

But do not install/rely on cygwin's outdated python environment.

Tip: To install *python*, you can try the free (as in beer) distribution Anaconda for *Windows* and *OS X*, or the totally free WinPython distribution, but only for *Windows*:

- For *Anaconda* you may need to install project's dependencies manually (see `setup.py`) using **`conda`**.
- The most recent version of *WinPython* (python-3.4) although it has just changed maintainer, it remains a higly active project, and it can even compile native libraries using an installations of *Visual Studio*, if available (required for instance when upgrading `numpy/scipy`, `pandas` or `matplotlib` with **`pip`**).

  You must also **Register your WinPython installation** and **add your installation into** `PATH` (see Frequently Asked Questions). To register it, go to *Start menu* → *All Programs* → *WinPython* → *WinPython ControlPanel*, and then *Options* → *Register Distribution* .

1.3 Discussion

CHAPTER 2

Install

Current 0.0.9-alpha.3 runs on Python-2.7+ and Python-3.3+ but 3.3+ is the preferred one, i.e, the desktop UI runs only with it. It is distributed on Wheels.

Before installing it, make sure that there are no older versions left over. So run this command until you cannot find any project installed:

```
$ pip uninstall wltp                          ## Use `pip3` if both python-2 & 3 are in PATH.
```

You can install the project directly from the *PyPi* repo the "standard" way, by typing the **`pip`** in the console:

```
$ pip install wltp  --pre
```

- If you want to install a *pre-release* version (the version-string is not plain numbers, but ends with `alpha`, `beta.2` or something else), use additionally `--pre`.
- If you want to upgrade an existing instalation along with all its dependencies, add also `--upgrade` (or `-U` equivalently), but then the build might take some considerable time to finish. Also there is the possibility the upgraded libraries might break existing programs(!) so use it with caution, or from within a *virtualenv* (isolated Python environment).
- To install it for different Python environments, repeat the procedure using the appropriate **`python.exe`** interpreter for each environment.
- **Tip:** To debug installation problems, you can export a non-empty `DISTUTILS_DEBUG` and *distutils* will print detailed information about what it is doing and/or print the whole command line when an external program (like a C compiler) fails.

After installation, it is important that you check which version is visible in your `PATH`:

```
$ wltp --version
0.0.9-alpha.3
```

To install for different Python versions, repeat the procedure for every required version.

2.1 Older versions

An additional purpose of the versioning schema of the project is to track which specific version of the GTR it implements. Given a version number `MAJOR.MINOR.PATCH`, the `MAJOR` part tracks the GTR phase implemented. See the "GTR version matrix" section in Changes for the mapping of MAJOR-numbers to GTR versions.

To install an older version issue the console command:

```
$ pip install wltp=1.1.1                      ## Use `--pre` if version-string has a build-suffix.
```

If you have another version already installed, you have to use `--ignore-installed` (or `-I`). For using the specific version, check this (untested) stackoverflow question .

Of course it is better to install each version in a separate *virtualenv* (isolated Python environment) and shy away from all this.

2.2 Installing from sources

If you download the sources you have more options for installation. There are various methods to get hold of them:

- Download the *source* distribution from *PyPi* repo.
- Download a release-snapshot from github
- Clone the *git-repository* at *github*.

  Assuming you have a working installation of git you can fetch and install the latest version of the project with the following series of commands:

```
$ git clone "https://github.com/ankostis/wltp.git" wltp.git
$ cd wltp.git
$ python setup.py install                                    ## Use `python3` if both python-2 &
```

When working with sources, you need to have installed all libraries that the project depends on:

```
$ pip install -r requirements/execution.txt .
```

The previous command installs a "snapshot" of the project as it is found in the sources. If you wish to link the project's sources with your python environment, install the project in development mode:

```
$ python setup.py develop
```

Note: This last command installs any missing dependencies inside the project-folder.

2.3 Project files and folders

The files and folders of the project are listed below:

```
+--wltp/             ## (package) The python-code of the calculator
|   +--cycles/       ## (package) The python-code for the WLTC data
|   +--test/         ## (package) Test-cases and the wltp_db
|   +--model         ## (module) Describes the data and their schema for the calculation
|   +--experiment    ## (module) The calculator
|   +--plots         ## (module) Diagram-plotting code and utilities
+--docs/             ## Documentation folder
|   +--pyplots/      ## (scripts) Plot the metric diagrams embeded in the README
+--devtools/         ## (scripts) Preprocessing of WLTC data on GTR and the wltp_db
|   +--run_tests.sh  ## (script) Executes all TestCases
+--wltp              ## (script) The cmd-line entry-point script for the calculator
+--setup.py          ## (script) The entry point for `setuptools`, installing, testing, etc
+--requirements/    ## (txt-files) Various pip-dependencies for tools.
+--README.rst
+--CHANGES.rst
+--LICENSE.txt
```

2.4 Discussion

CHAPTER 3

Usage

3.1 Cmd-line usage

Warning: Not implemented in yet.

The command-line usage below requires the Python environment to be installed, and provides for executing an experiment directly from the OS's shell (i.e. **cmd** in windows or **bash** in POSIX), and in a *single* command. To have precise control over the inputs and outputs (i.e. experiments in a "batch" and/or in a design of experiments) you have to run the experiments using the API python, as explained below.

The entry-point script is called **wltp**, and it must have been placed in your `PATH` during installation. This script can construct a *model* by reading input-data from multiple files and/or overriding specific single-value items. Conversely, it can output multiple parts of the resulting-model into files.

To get help for this script, use the following commands:

```
$ wltp --help                                ## to get generic help for cmd-line syntax
$ wltcmdp.py -M vehicle/full_load_curve      ## to get help for specific model-paths
```

and then, assuming `vehicle.csv` is a CSV file with the vehicle parameters for which you want to override the `n_idle` only, run the following:

```
$ wltp -v \
    -I vehicle.csv file_frmt=SERIES model_path=params header@=None \
    -m vehicle/n_idle:=850 \
    -O cycle.csv model_path=cycle_run
```

3.2 GUI usage

Attention: Desktop UI requires Python 3!

For a quick-'n-dirty method to explore the structure of the model-tree and run an experiment, just run:

```
$ wltp --gui
```

3.3 Excel usage

Attention: Excel-integration requires Python 3 and *Windows* or *OS X*!

In *Windows* and *OS X* you may utilize the excellent xlwings library to use Excel files for providing input and output to the experiment.

To create the necessary template-files in your current-directory you should enter:

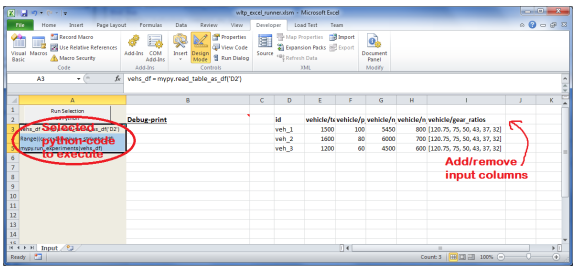
```
$ wltp --excel
```

You could type instead `wltp --excel` *`file_path`* to specify a different destination path.

In *windows*/*OS X* you can type `wltp --excelrun` and the files will be created in your home-directory and the excel will open them in one-shot.

All the above commands creates two files:

`wltp_excel_runner.xlsm` The python-enabled excel-file where input and output data are written, as seen in the screenshot below:



After opening it the first tie, enable the macros on the workbook, select the python-code at the left and click the *Run Selection as Pyhon* button; one sheet per vehicle should be created.

The excel-file contains additionally appropriate *VBA* modules allowing you to invoke *Python code* present in *selected cells* with a click of a button, and python-functions declared in the python-script, below, using the `mypy` namespace.

To add more input-columns, you need to set as column *Headers* the *json-pointers* path of the desired model item (see *Python usage* below,).

`wltp_excel_runner.py` Utility python functions used by the above xls-file for running a batch of experiments.

The particular functions included reads multiple vehicles from the input table with various vehicle characteristics and/or experiment parameters, and then it adds a new worksheet containing the cycle-run of each vehicle . Of course you can edit it to further fit your needs.

Note: You may reverse the procedure described above and run the python-script instead. The script will open the excel-file, run the experiments and add the new sheets, but in case any errors occur, this time you can debug them, if you had executed the script through *LiClipse*, or *IPython*!

Some general notes regarding the python-code from excel-cells:

- On each invocation, the predefined VBA module `pandalon` executes a dynamically generated python-script file in the same folder where the excel-file resides, which, among others, imports the "sister" python-script file. You can read & modify the sister python-script to import libraries such as 'numpy' and 'pandas', or pre-define utility python functions.
- The name of the sister python-script is automatically calculated from the name of the Excel-file, and it must be valid as a python module-name. Therefore do not use non-alphanumeric characters such as spaces('), `dashes` (-) `and dots` (.') on the Excel-file.
- On errors, a log-file is written in the same folder where the excel-file resides, for as long as **the message-box is visible, and it is deleted automatically after you click 'ok'!**
- Read http://docs.xlwings.org/quickstart.html

3.4 Python usage

Example python REPL (Read-Eval-Print Loop) example-commands are given below that setup and run an *experiment*.

First run **`python`** or **`ipython`** and try to import the project to check its version:

```
>>> import wltp

>>> wltp.__version__              ## Check version once more.
'0.0.9-alpha.3'

>>> wltp.__file__                 ## To check where it was installed.
/usr/local/lib/site-package/wltp-...
```

If everything works, create the *pandas-model* that will hold the input-data (strings and numbers) of the experiment. You can assemble the model-tree by the use of:

- sequences,
- dictionaries,
- `pandas.DataFrame`,
- `pandas.Series`, and
- URI-references to other model-trees.

For instance:

```
>>> from wltp import model
>>> from wltp.experiment import Experiment
>>> from collections import OrderedDict as odic          ## It is handy to preserve keys-order.

>>> mdl = odic(
...   vehicle = odic(
...     unladen_mass = 1430,
...     test_mass    = 1500,
...     v_max        = 195,
...     p_rated      = 100,
...     n_rated      = 5450,
...     n_idle       = 950,
...     n_min        = None,                              ## Manufacturers my overridde it
...     gear_ratios         = [120.5, 75, 50, 43, 37, 32],
...     resistance_coeffs   = [100, 0.5, 0.04],
...   )
... )
```

For information on the accepted model-data, check its *JSON-schema*:

```
>>> model.json_dumps(model.model_schema(), indent=2)
{
  "properties": {
    "params": {
      "properties": {
        "f_n_min_gear2": {
          "description": "Gear-2 is invalid when N :< f_n_min_gear2 * n_idle.",
          "type": [
            "number",
            "null"
          ],
          "default": 0.9
        },
        "v_stopped_threshold": {
          "description": "Velocity (Km/h) under which (<=) to idle gear-shift (Annex 2-3.3, p71).
```

```
          "type": [
...
```

You then have to feed this model-tree to the *Experiment* constructor. Internally the *Pandel* resolves URIs, fills-in default values and validates the data based on the project's pre-defined JSON-schema:

```
>>> processor = Experiment(mdl)          ## Fills-in defaults and Validates model.
```

Assuming validation passes without errors, you can now inspect the defaulted-model before running the experiment:

```
>>> mdl = processor.model                ## Returns the validated model with filled-in defaults.
>>> sorted(mdl)                          ## The "defaulted" model now includes the `params` branch
['params', 'vehicle']
>>> 'full_load_curve' in mdl['vehicle'] ## A default wot was also provided in the `vehicle`.
True
```

Now you can run the experiment:

```
>>> mdl = processor.run()                ## Runs experiment and augments the model with results.
>>> sorted(mdl)                          ## Print the top-branches of the "augmented" model.
['cycle_run', 'params', 'vehicle']
```

To access the time-based cycle-results it is better to use a `pandas.DataFrame`:

```
>>> import pandas as pd
>>> df = pd.DataFrame(mdl['cycle_run']); df.index.name = 't'
>>> df.shape                             ## ROWS(time-steps) X COLUMNS.
(1801, 11)
>>> df.columns
Index(['v_class', 'v_target', 'clutch', 'gears_orig', 'gears', 'v_real', 'p_available', 'p_require
>>> 'Mean engine_speed: %s' % df.rpm.mean()
'Mean engine_speed: 1917.0407829'
>>> df.describe()
           v_class     v_target     clutch   gears_orig        gears  \
count  1801.000000  1801.000000       1801  1801.000000  1801.000000
mean     46.506718    46.506718  0.0660744     3.794003     3.683509
std      36.119280    36.119280  0.2484811     2.278959     2.278108
...

            v_real  p_available   p_required          rpm     rpm_norm
count  1801.000000  1801.000000  1801.000000  1801.000000  1801.000000
mean     50.356222    28.846639     4.991915  1917.040783     0.214898
std      32.336908    15.833262    12.139823   878.139758     0.195142
...

>>> processor.driveability_report()
...
  12: (a: X-->0)
  13: g1: Revolutions too low!
  14: g1: Revolutions too low!
...
  30: (b2(2): 5-->4)
...
  38: (c1: 4-->3)
  39: (c1: 4-->3)
  40: Rule e or g missed downshift(40: 4-->3) in acceleration?
...
  42: Rule e or g missed downshift(42: 3-->2) in acceleration?
...
```

You can export the cycle-run results in a CSV-file with the following pandas command:

```
>>> df.to_csv('cycle_run.csv')
```

For more examples, download the sources and check the test-cases found under the `/wltp/test/` folder.

3.5 IPython notebook usage

The list of *IPython notebooks* for wltp is maintained at the wiki of the project.

3.5.1 Requirements

In order to run them interactively, ensure that the following requirements are satisfied:

1. A ipython-notebook server >= v2.x.x is installed for *python-3*, it is up, and running.
2. The *wltp* is installed on your system (see Install above).

3.5.2 Instructions

- Visit each *notebook* from the wiki-list that you wish to run and **download** it as `ipynb` file from the menu (*File\|Download as...\|IPython Notebook(.ipynb)*).
- Locate the downloaded file with your *file-browser* and **drag n' drop** it on the landing page of your notebook's server (the one with the folder-list).

Enjoy!

3.6 Discussion

CHAPTER 4

Getting Involved

This project is hosted in **github**. To provide feedback about bugs and errors or questions and requests for enhancements, use github's Issue-tracker.

4.1 Sources & Dependencies

To get involved with development, you need a POSIX environment to fully build it (*Linux*, *OSX* or *Cygwin* on *Windows*).

First you need to download the latest sources:

```
$ git clone https://github.com/ankostis/wltp.git wltp.git
$ cd wltp.git
```

Virtualenv

You may choose to work in a *virtualenv* (isolated Python environment), to install dependency libraries isolated from system's ones, and/or without *admin-rights* (this is recommended for *Linux*/*Mac OS*).

> **Attention:** If you decide to reuse stystem-installed packages using `--system-site-packages` with `virtualenv <= 1.11.6` (to avoid, for instance, having to reinstall *numpy* and *pandas* that require native-libraries) you may be bitten by bug #461 which prevents you from upgrading any of the pre-installed packages with **`pip`**.

Liclipse IDE

Within the sources there are two sample files for the comprehensive LiClipse IDE:

- `eclipse.project`
- `eclipse.pydevproject`

Remove the `eclipse` prefix, (but leave the dot()) and import it as "existing project" from Eclipse's `File` menu.

Another issue is caused due to the fact that LiClipse contains its own implementation of *Git*, *EGit*, which badly interacts with unix *symbolic-links*, such as the `docs/docs`, and it detects working-directory changes even after a fresh checkout. To workaround this, Right-click on the above file *Properties* → *Team* → *Advanced* → *Assume Unchanged*

Then you can install all project's dependencies in *'development mode* using the `setup.py` script:

```
$ python setup.py --help                          ## Get help for this script.
Common commands: (see '--help-commands' for more)
```

```
  setup.py build      will build the package underneath 'build/'
  setup.py install    will install the package

Global options:
  --verbose (-v)      run verbosely (default)
  --quiet (-q)        run quietly (turns verbosity off)
  --dry-run (-n)      don't actually do anything
...

$ python setup.py develop                          ## Also installs dependencies into project's
$ python setup.py build                            ## Check that the project indeed builds ok.
```

You should now run the test-cases (see ref:`metrics`, below) to check that the sources are in good shape:

```
$ python setup.py test
```

Note: The above commands installed the dependencies inside the project folder and for the *virtual-environment*. That is why all build and testing actions have to go through `python setup.py some_cmd`.

If you are dealing with installation problems and/or you want to permantly install dependant packages, you have to *deactivate* the virtual-environment and start installing them into your *base* python environment:

```
$ deactivate
$ python setup.py develop
```

or even try the more *permanent* installation-mode:

```
$ python setup.py install                # May require admin-rights
```

4.2 Development procedure

For submitting code, use `UTF-8` everywhere, unix-eol(`LF`) and set `git --config core.autocrlf = input`.

The typical development procedure is like this:

1. Modify the sources in small, isolated and well-defined changes, i.e. adding a single feature, or fixing a specific bug.
2. Add test-cases "proving" your code.
3. Rerun all test-cases to ensure that you didn't break anything, and check their *coverage* remain above 80%:

```
        $ python setup.py nosetests --with-coverage --cover-package wltp.model,wltp.experiment --
```

> **Tip:** You can enter just: `python setup.py test_all` instead of the above cmd-line since it has been *aliased* in the `setup.cfg` file. Check this file for more example commands to use during development.

4. If you made a rather important modification, update also the Changes file and/or other documents (i.e. README.rst). To see the rendered results of the documents, issue the following commands and read the result html at `build/sphinx/html/index.html`:

```
        $ python setup.py build_sphinx                  # Builds html docs
        $ python setup.py build_sphinx -b doctest       # Checks if python-code embeded in commen
```

5. If there are no problems, commit your changes with a descriptive message.
6. Repeat this cycle for other bugs/enhancements.

7. When you are finished, push the changes upstream to *github* and make a *merge_request*. You can check whether your merge-request indeed passed the tests by checking its build-status on the integration-server's site (TravisCI).

> **Hint:** Skim through the small IPython developer's documentantion on the matter: The perfect pull request

4.3 Specs & Algorithm

This program was implemented from scratch based on this `GTR specification` (included in the `docs/` folder). The latest version of this GTR, along with other related documents can be found at UNECE's site:

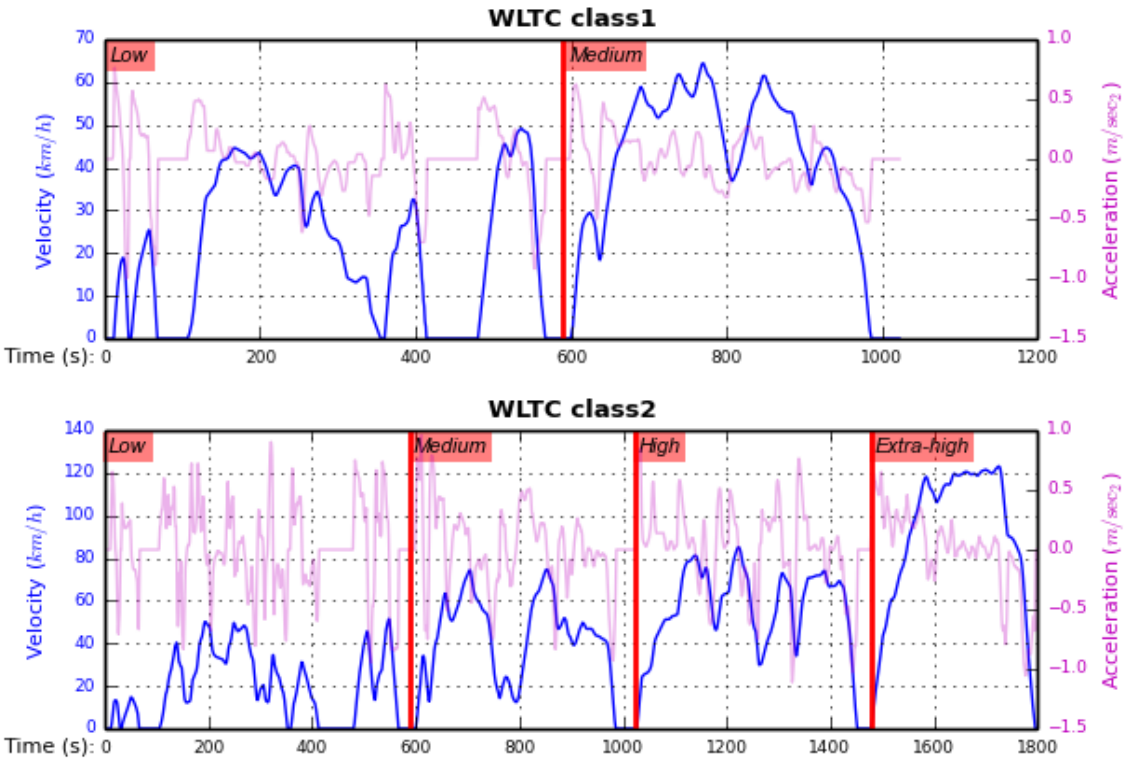
- http://www.unece.org/trans/main/wp29/wp29wgs/wp29grpe/grpedoc_2013.html
- https://www2.unece.org/wiki/pages/viewpage.action?pageId=2523179
- Probably a more comprehensible but older spec is this one: https://www2.unece.org/wiki/display/trans/DHC+draft+technical+

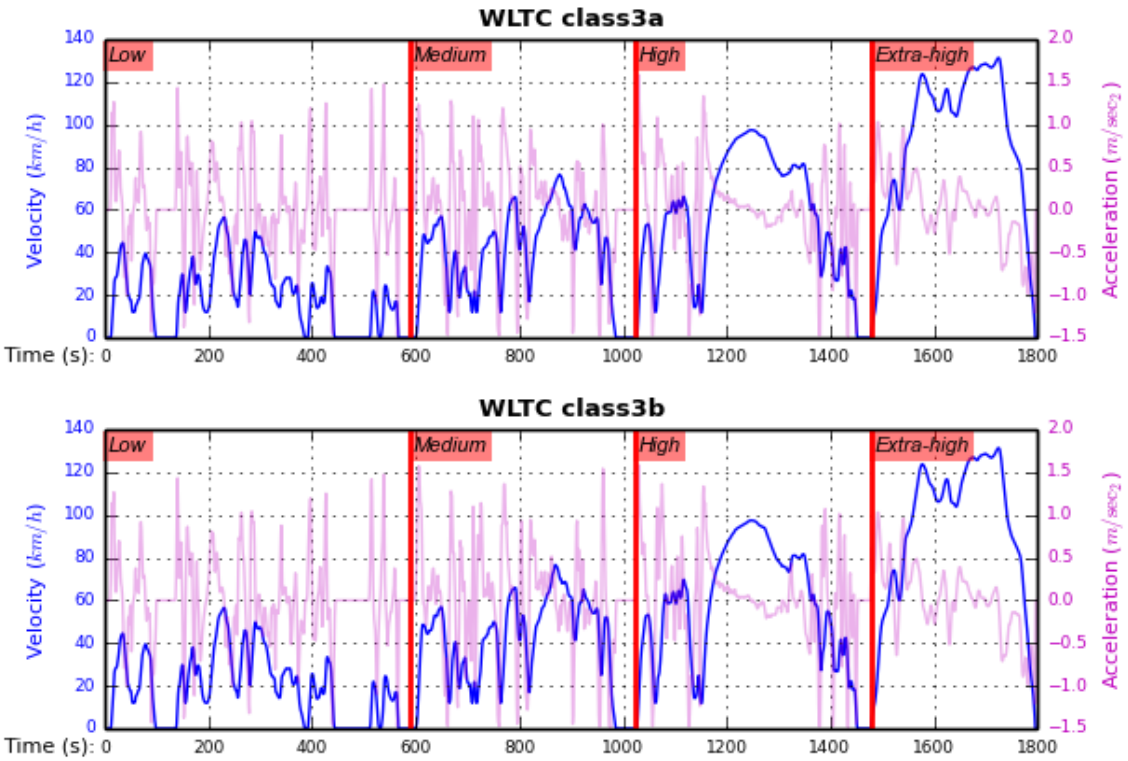
The WLTC-profiles for the various classes in the `devtools/data/cycles/` folder were generated from the tables of the specs above using the `devtools/csvcolumns8to2.py` script, but it still requires an intermediate manual step involving a spreadsheet to copy the table into ands save them as CSV.

Then use the `devtools/buildwltcclass.py` to construct the respective python-vars into the `wltp/model.py` sources.

Data-files generated from Steven Heinz's ms-access `vehicle info` db-table can be processed with the `devtools/preprocheinz.py` script.

4.3.1 Cycles





4.4 Tests, Metrics & Reports

In order to maintain the algorithm stable, a lot of effort has been put to setup a series of test-case and metrics to check the sanity of the results and to compare them with the Heinz-db tool or other datasets included in the project. These tests can be found in the `wltp/test/` folders.

Additionally, below are *auto-generated* representative diagrams with the purpose to track the behavior and the evolution of this project.

You can reuse the plotting code here for building nice ipython-notebooks reports, and (optionally) link them in the wiki of the project (see section above). The actual code for generating diagrams for these metrics is in `wltp.plots` and it is invoked by scripts in the `docs/pyplot/` folder.

4.4.1 *Mean Engine-speed* vs *PMR*

First the mean engine-speed of vehicles are compared with access-db tool, grouped by PMRs:

Both tools generate the same rough engine speeds. There is though a trend for this project to produce lower rpm's as the PMR of the vehicle increases. But it is difficult to tell what each vehicle does isolated.

The same information is presented again but now each vehicle difference is drawn with an arrow:

It can be seen now that this project's calculates lower engine-speeds for classes 1 & 3 but the trend is reversed for class 2.

4.4.2 *Mean Engine-speed* vs *Gears*

Below the mean-engine-speeds are drawn against the mean gear used, grouped by classes and class-parts (so that, for instance, a class3 vehicle corresponds to 3 points on the diagram):

4.5 Development team

- **Author:**
  - Kostis Anagnostopoulos
- **Contributing Authors:**
  - Heinz Steven (test-data, validation and review)
  - Georgios Fontaras (simulation, physics & engineering support)
  - Alessandro Marotta (policy support)

4.6 Discussion

CHAPTER 5

Frequently Asked Questions

5.1 General

5.1.1 Who is behind this? Who to contact?

The immediate involved persons is described in the *Development team* section. The author is a participating member in the *GS Task-Force* on behalf of the EU Commission (JRC). The contact-emails to use are ...[TBD]

5.1.2 What is the status of the project? Is it "official"?

[TBD]

5.1.3 What is the roadmap for this project?

- Short-term plans are described in the *TODOs* section of Changes.
- In the longer run, it is expected to incorporate more *WLTP* calculations and reference data so that this projects acts as repository for diagrams and technical reports on those algorithms.

5.1.4 Can I copy/extend it? What is its License, in practical terms?

I'm not a lawyer, but in a broad view, the core algorithm of the project is "copylefted" with the *EUPL-1.1+ license*, and it includes files from other "non-copyleft" open source licenses like *MIT MIT License* and *Apache License*, appropriately marked as such. So in an nutshell, you can study it, copy it, modify or extend it, and distrbute it, as long as you always distribute the sources of your changes.

5.2 Technical

5.2.1 I followed the instructions but i still cannot install/run/do *X*. What now?

If you have no previous experience in python, setting up your environment and installing a new project is a demanding, but manageable, task. Here is a checklist of things that might go wrong:

- Did you send each command to the **appropriate shell/interpreter**?

  You should enter sample commands starting `$` into your *shell* (**`cmd`** or **`bash`**), and those starting with `>>>` into the *python-interpreter* (but don't include the previous symbols and/or the *output* of the commands).

- Is **python contained in your PATH** ?

  To check it, type `python` in your console/command-shell prompt and press `[Enter]`. If nothing happens, you have to inspect `PATH` and modify it accordingly to include your python-installation.

  - Under *Windows* type `path` in your command-shell prompt. To change it, run **`regedit.exe`** and modify (or add if not already there) the `PATH` string-value inside the following *registry-setting*:

```
HKEY_CURRENT_USER\Environment\
```

    You need to logoff and logon to see the changes.

    Note that *WinPython* **does not modify your path!** if you have registed it, so you definetely have to perform the the above procedure yourself.

  - Under *Unix* type `echo $PATH$` in your console. To change it, modify your "rc' files, ie: `~/.bashrc` or `~/.profile`.

- Is the correct **version of python** running?

  Certain commands such as **`pip`** come in 2 different versions *python-2 & 3* (**`pip2`** and **`pip3`**, respectively). Most programs report their version-infos with `--version`. Use `--help` if this does not work.

- Have you **upgraded/downgraded the project** into a more recent/older version?

  This project is still in development, so the names of data and functions often differ from version to version. Check the Changes for point that you have to be aware of when upgrading.

- Did you search whether **a similar issue** has already been reported?

- Did you **ask google** for an answer??

- If the above suggestions still do not work, feel free to **open a new issue** and ask for help. Write down your platform (Windows, OS X, Linux), your exact python distribution and version, and include the *print-out of the failed command along with its error-message.*

  This last step will improve the documentation and help others as well.

5.2.2 I do not have python / cannot install it. Is it possible to try a *demo*?

[TBD]

5.2.3 Discussion

CHAPTER 6

API reference

The core of the simulator is composed from the following modules:

pandel	A *pandas-model* is a tree of strings, numbers, sequences, dicts, pandas instances and resolvable URI-references,
model	Defines the schema, defaults and validation operations for the data consumed and produced by the *Experimen*
experiment	The core that accepts a vehicle-model and wltc-classes, runs the simulation and updates the model with results (

Among the various tests, those running on 'sample' databases for comparing differences with existing tool are the following:

samples_db_tests	Compares the results of synthetic vehicles from JRC against pre-phase-1b Heinz's tool.
wltp_db_tests	Compares the results of a batch of wltp_db vehicles against phase-1b-alpha Heinz's tool.

The following scripts in the sources maybe used to preprocess various wltc data:

- `devtools/preprocheinz.py`
- `devtools/printwltcclass.py`
- `devtools/csvcolumns8to2.py`

6.1 Module: `wltp.experiment`

The core that accepts a vehicle-model and wltc-classes, runs the simulation and updates the model with results (downscaled velocity & gears-profile).

Attention: The documentation of this core module has several issues and needs work.

6.1.1 Notation

- ALL_CAPITAL variables denote *vectors* over the velocity-profile (the cycle),
- ALL_CAPITAL starting with underscore (_) denote *matrices* (gears x time).

For instance, GEARS is like that:

```
[0, 0, 1, 1, 1, 2, 2, ... 1, 0, 0]
 <----   cycle time-steps   ---->
```

and _GEARS is like that:

```
 t:||: 0  1  2  3
---+-------------
g1:|[[ 1, 1, 1, 1, ... 1, 1
g2:|   2, 2, 2, 2, ... 2, 2
g3:|   3, 3, 3, 3, ... 3, 3
g4:|   4, 4, 4, 4, ... 4, 4 ]]
```

6.1.2 Major vectors & matrices

V: floats (#cycle_steps) The wltp-class velocity profile.

_GEARS: integers (#gears X #cycle_steps) One row for each gear (starting with 1 to #gears).

_N_GEARS: floats (#gears X #cycle_steps) One row per gear with the Engine-revolutions required to follow the V-profile (unfeasable revs included), produced by multiplying `V * gear-rations`.

_GEARS_YES: boolean (#gears X #cycle_steps) One row per gear having `True` wherever gear is possible for each step.

See also:

`model` for in/out schemas

class `wltp.experiment.`**Experiment**(*model*, *skip_model_validation=False*, *validate_wltc_data=False*)

Bases: `object`

Runs the vehicle and cycle data describing a WLTC experiment.

See *`wltp.experiment`* for documentation.

__init__(*model*, *skip_model_validation=False*, *validate_wltc_data=False*)

Parameters

- **model** – trees (formed by dicts & lists) holding the experiment data.
- **skip_model_validation** – when true, does not validate the model.

run()

Invokes the main-calculations and extracts/update Model values!

@see: Annex 2, p 70

`wltp.experiment.`**applyDriveabilityRules**(*V*, *A*, *GEARS*, *CLUTCH*, *driveability_issues*)

@note: Modifies GEARS & CLUTCH. @see: Annex 2-4, p 72

`wltp.experiment.`**calcDownscaleFactor**(*P_REQ*, *p_max_values*, *downsc_coeffs*, *dsc_v_split*, *p_rated*, *v_max*, *f_downscale_threshold*)

Check if downscaling required, and apply it.

Returns (float) the factor

@see: Annex 1-7, p 68

`wltp.experiment.`**calcEngineRevs_required**(*V*, *gear_ratios*, *n_idle*, *v_stopped_threshold*)

Calculates the required engine-revolutions to achieve target-velocity for all gears.

Returns array: _N_GEARS: a (#gears X #velocity) float-array, eg. [3, 150] –> gear(3), time(150)

Return type array: _GEARS: a (#gears X #velocity) int-array, eg. [3, 150] –> gear(3), time(150)

@see: Annex 2-3.2, p 71

`wltp.experiment.`**calcPower_available**(*_N_GEARS*, *n_idle*, *n_rated*, *p_rated*, *load_curve*, *p_safety_margin*)

@see: Annex 2-3.2, p 72

wltp.experiment.**calcPower_required**(*V*, *A*, *SLOPE*, *test_mass*, *f0*, *f1*, *f2*, *f_inertial*)
@see: Annex 2-3.1, p 71

wltp.experiment.**decideClass**(*wltc_data*, *p_m_ratio*, *v_max*)
@see: Annex 1, p 19

wltp.experiment.**downscaleCycle**(*V*, *f_downscale*, *phases*)
Downscale just by scaling the 2 phases demarked by the 3 time-points with different factors, no recursion as implied by the specs.

@see: Annex 1-7, p 64-68

wltp.experiment.**gearsregex**(*gearspattern*)

Parameters **gearspattern** – regular-expression or substitution that escapes decimal-bytes written as: `\g\d+` with adding +128, eg:

```
\g124|\g7 --> unicode(128+124=252)|unicode(128+7=135)
```

wltp.experiment.**possibleGears_byEngineRevs**(*V*, *A*, *_N_GEARS*, *ngears*, *n_idle*, *n_min_drive*, *n_min_gear2*, *n_max*, *v_stopped_threshold*, *driveability_issues*)
Calculates the engine-revolutions limits for all gears and returns for which they are accepted.

My interpratation for Gear2 `n_min` limit:

```
                      ///INVALID///|   CLUTCHED   |  GEAR-2-OK
EngineRevs(N): 0-----------------------+---------------------------->
for Gear-2                     |       |      +--> n_clutch_gear2   := n_idle + MAX(
                               |       |                                0.15% * n_idle,
                               |       |                                3%    * n_range
                               |       +---------> n_idle
                               +-----------------> n_min_gear2      := 90% * n_idle
```

Returns _GEARS_YES: possibibilty for all the gears on each cycle-step (eg: [0, 10] == True –> gear(1) is possible for t=10)

Return type list(booleans, nGears x CycleSteps)

@see: Annex 2-3.2, p 71

wltp.experiment.**possibleGears_byPower**(*_N_GEARS*, *P_REQ*, *n_idle*, *n_rated*, *p_rated*, *load_curve*, *p_safety_margin*, *driveability_issues*)
@see: Annex 2-3.1 & 3.3, p 71 & 72

wltp.experiment.**rule_a**(*bV*, *GEARS*, *CLUTCH*, *driveability_issues*, *re_zeros*)
Rule (a): Clutch & set to 1st-gear before accelerating from standstill.

Implemented with a regex, outside rules-loop: Also ensures gear-0 always followed by gear-1.

NOTE: Rule(A) not inside x2 loop, and last to run.

wltp.experiment.**rule_c2**(*bV*, *A*, *GEARS*, *CLUTCH*, *driveability_issues*, *re_zeros*)
Rule (c2): Skip 1st-gear while decelerating to standstill.

Implemented with a regex, outside rules-loop: Search for zeros in _reversed_ V & GEAR profiles, for as long Accel is negative. NOTE: Rule(c2) is the last rule to run.

wltp.experiment.**run_cycle**(*V*, *A*, *P_REQ*, *gear_ratios*, *n_idle*, *n_min_drive*, *n_rated*, *p_rated*, *load_curve*, *params*)
Calculates gears, clutch and actual-velocity for the cycle (V). Initial calculations happen on engine_revs for all gears, for all time-steps of the cycle (_N_GEARS array). Driveability-rules are applied afterwards on the selected gear-sequence, for all steps.

Parameters

- **V** – the cycle, the velocity profile
- **A** – acceleration of the cycle (diff over V) in m/sec^2

Returns CLUTCH: a (1 X #velocity) bool-array, eg. [3, 150] –> gear(3), time(150)

Return type array

wltp.experiment.**step_rule_b1**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule (b1): Do not skip gears while accelerating.

wltp.experiment.**step_rule_b2**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule (b2): Hold gears for at least 3sec when accelerating.

wltp.experiment.**step_rule_c1**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule (c1): Skip gears <3sec when decelerating.

wltp.experiment.**step_rule_d**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule (d): Cancel shifts after peak velocity.

wltp.experiment.**step_rule_e**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule (e): Cancel shifts lasting 5secs or less.

wltp.experiment.**step_rule_f**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule(f): Cancel 1sec downshifts (under certain circumstances).

wltp.experiment.**step_rule_g**(*t*, *pg*, *g*, *V*, *A*, *GEARS*, *driveability_issues*)
Rule(g): Cancel upshift during acceleration if later downshifted for at least 2sec.

6.2 Module: wltp.model

Defines the schema, defaults and validation operations for the data consumed and produced by the *Experiment*.

The model-instance is managed by pandel.Pandel.

wltp.model.**_get_model_base**()
The base model for running a WLTC experiment.

It contains some default values for the experiment (ie the default 'full-load-curve' for the vehicles). But note that it this model is not valid - you need to override its attributes.

Returns a tree with the default values for the experiment.

wltp.model.**_get_model_schema**(*additional_properties=False*, *for_prevalidation=False*)

Parameters **additional_properties** (*bool*) – when False, 4rd-step(validation) will scream on any non-schema property found.

Returns The json-schema(dict) for input/output of the WLTC experiment.

wltp.model.**_get_wltc_data**()
The WLTC-data required to run an experiment (the class-cycles and their attributes)..

Prefer to access wltc-data through *model['wltc_data']*.

Returns a tree

wltp.model.**_get_wltc_schema**()
The json-schema for the WLTC-data required to run a WLTC experiment.

:return :dict:

wltp.model.**get_class_part_names**(*cls_name=None*)

Parameters **cls_name** (*str*) – one of 'class1', ..., 'class3b', if missing, returns all 4 part-names

wltp.model.**get_class_parts_limits**(*cls_name*, *mdl=None*, *edges=False*)
Parses the supplied in wltc_data and extracts the part-limits for the specified class-name.

Parameters

- **cls_name** (*str*) – one of 'class1', ..., 'class3b'
- **mdl** – the mdl to parse wltc_data from, if ommited, parses the results of *_get_wltc_data()*
- **edges** – when `True`, embeds internal limits into (0, len)

Returns a list with the part-limits, ie for class-3a these are 3 numbers

`wltp.model.`**get_class_pmr_limits**(*mdl=None*, *edges=False*)

Parses the supplied in wltc_data and extracts the part-limits for the specified class-name.

Parameters

- **mdl** – the mdl to parse wltc_data from, if omitted, parses the results of *_get_wltc_data()*
- **edges** – when `True`, embeds internal limits into (0, len)

Returns a list with the pmr-limits (2 numbers)

`wltp.model.`**get_model_schema**(*additional_properties=False*, *for_prevalidation=False*)

Parameters **additional_properties** (*bool*) – when False, 4rd-step(validation) will scream on any non-schema property found.

Returns The json-schema(dict) for input/output of the WLTC experiment.

`wltp.model.`**merge**(*a*, *b*, *path=[]*)

'merges b into a

`wltp.model.`**validate_model**(*mdl*, *additional_properties=False*, *iter_errors=False*, *validate_wltc_data=False*, *validate_schema=False*)

Parameters **iter_errors** (*bool*) – does not fail, but returns a generator of ValidationErrors

```
>>> validate_model(None)
Traceback (most recent call last):
jsonschema.exceptions.ValidationError: None is not of type 'object'
...
```

```
>>> mdl = _get_model_base()
>>> err_generator = validate_model(mdl, iter_errors=True)
>>> sorted(err_generator, key=hash)
[<ValidationError:
...
```

```
>>> mdl = _get_model_base()
>>> mdl["vehicle"].update({
...     "unladen_mass":1230,
...     "test_mass":   1300,
...     "v_max":   195,
...     "p_rated": 110.625,
...     "n_rated": 5450,
...     "n_idle":  950,
...     "n_min":   500,
...     "gear_ratios":[120.5, 75, 50, 43, 33, 28],
...     "resistance_coeffs":[100, 0.5, 0.04],
... })
>>> err_generator = validate_model(mdl, iter_errors=True)
>>> len(list(err_generator))
0
```

6.3 Module: wltp.pandel

A *pandas-model* is a tree of strings, numbers, sequences, dicts, pandas instances and resolvable URI-references, implemented by *Pandel*.

class wltp.pandel.**ModelOperations**
Bases: *wltp.pandel.ModelOperations*

Customization functions for traversing, I/O, and converting self-or-descendant branch (sub)model values.

static __new__(*inp=None*, *out=None*, *conv=None*)

Parameters

- **inp** (*list*) – the `args-list` to *Pandel._read_branch()*
- **out** – The args to *Pandel._write_branch()*, that may be specified either as:
  - an `args-list`, that will apply for all model data-types (lists, dicts & pandas),
  - a map of `type` –> `args-list`, where the `None` key is the *catch-all* case,
  - a function returning the `args-list` for some branch-value, with signature: `def get_write_branch_args(branch)`.
- **conv** – The conversion-functions (*convertors*) for the various model's data-types. The convertors have signature `def convert(branch)`, and they may be specified either as:
  - a map of `(from_type, to_type)` –> `conversion_func()`, where the `None` key is the *catch-all* case,
  - a "master-switch" function returning the appropriate convertor depending on the requested conversion. The master-function's signature is `def get_convertor(from_branch, to_branch)`.

  The minimum convertors demanded by *Pandel* are (at least, check the code for more):
  - DataFrame <–> dict
  - Series <–> dict
  - ndarray <–> list

class wltp.pandel.**Pandel**(*curate_funcs=()*)
Bases: `object`

Builds, validates and stores a *pandas-model*, a mergeable stack of JSON-schema abiding trees of strings and numbers, assembled with

•sequences,

•dictionaries,

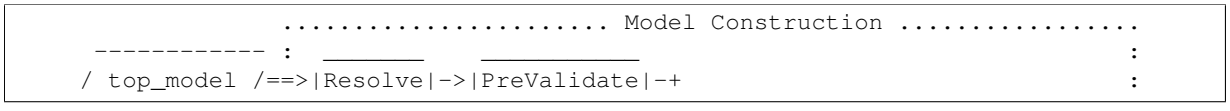
•`pandas.DataFrame`,

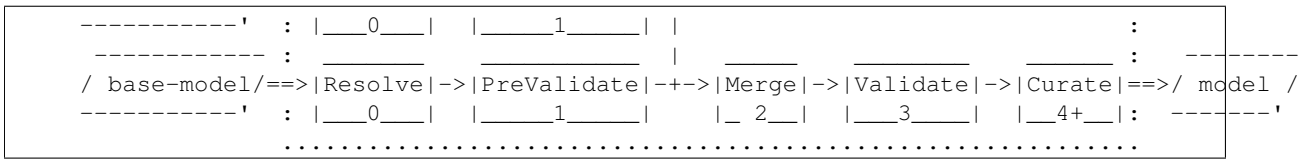
•`pandas.Series`, and

•URI-references to other model-trees.

Overview

The **making of a model** involves, among others, schema-validating, reading *subtree-branches* from URIs, cloning, converting and merging multiple *sub-models* in a single *unified-model* tree, without side-effecting given input. All these happen in 4+1 steps:





All steps are executed "lazily" using generators (with `yield`). Before proceeding to the next step, the previous one must have completed successfully. That way, any ad-hoc code in building-step-5(*curation*), for instance, will not suffer a horrible death due to badly-formed data.

[TODO] The **storing of a model** simply involves distributing model parts into different files and/or formats, again without side-effecting the unified-model. **Building model**

Here is a detailed description of each building-step:

1. *`_resolve()`* and substitute any json-references present in the submodels with content-fragments fetched from the referred URIs. The submodels are **cloned** first, to avoid side-effecting them.

   Although by default a combination of *JSON* and *CSV* files is expected, this can be customized, either by the content in the json-ref, within the model (see below), or as *explained* below.

   The **extended json-refs syntax** supported provides for passing arguments into *`_read_branch()`* and *`_write_branch()`* methods. The syntax is easier to explain by showing what the default *`_global_cntxt`* corresponds to, for a `DataFrame`:

```
{
  "$ref": "http://example.com/example.json#/foo/bar",
  "$inp": ["AUTO"],
  "$out": ["CSV", "encoding=UTF-8"]
}
```

   And here what is required to read and (later) store into a HDF5 local file with a predefined name:

```
{
  "$ref": "file://./filename.hdf5",
  "$inp": ["AUTO"],
  "$out": ["HDF5"]
}
```

   Warning: Step NOT IMPLEMENTED YET!

2. Loosely *`_prevalidate()`* each sub-model separately with *json-schema*, where any pandas-instances (DataFrames and Series) are left as is. It is the duty of the developer to ensure that the prevalidation-schema is *loose enough* that it allows for various submodel-forms, prior to merging, to pass.

3. Recursively **clone** and *`_merge()`* sub-models in a single unified-model tree. Branches from sub-models higher in the stack override the respective ones from the sub-models below, recursively. Different object types need to be **converted** appropriately (ie. merging a `dict` with a `DataFrame` results into a `DataFrame`, so the dictionary has to convert to dataframe).

   The required **conversions** into pandas classes can be customized as *explained* below. Series and DataFrames cannot merge together, and Sequences do not merge with any other object-type (themselfs included), they just "overwrite".

   The default convertor-functions defined both for submodels and models are listed in the following table:

From:	To:	Method:
dict	DataFrame	`pd.DataFrame` (the constructor)
DataFrame	dict	`lambda df: df.to_dict('list')`
dict	Series	`pd.Series` (the constructor)
Series	dict	`lambda sr: sr.to_dict()`

4. Strictly json-*_validate()* the unified-model (ie enforcing `required` schema-rules).

   The required **conversions** from pandas classes can be customized as *explained* below.

   The default convertor-functions are the same as above.

5. (Optionally) Apply the *_curate()* functions on the the model to enforce dependencies and/or any ad-hoc generation-rules among the data. You can think of bash-like expansion patterns, like `${/some/path:=$HOME}` or expressions like `%len(../other/path)`.

Storing model

When storing model-parts, if unspecified, the filenames to write into will be deduced from the jsonpointer-path of the `$out`'s parent, by substituting "strange" chars with undescores(_).

> **Warning:** Functionality NOT IMPLEMENTED YET!

Customization

Some operations within steps (namely *conversion* and *IO*) can be customized by the following means (from lower to higher precedance):

1. The global-default *ModelOperations* instance on the *_global_cntxt*, applied on both sub-models and unified-model.

   For example to channel the whole reading/writing of models through HDF5 data-format, it would suffice to modify the *_global_cntxt* like that:

```
pm = FooPandelModel()                          ## some concrete model-maker
io_args = ["HDF5"]
pm.mod_global_operations(inp=io_args, out=io_args)
```

2. [TODO] Extra-properties on the json-schema applied on both submodels and unified-model for the specific path defined. The supported properties are the non-functional properties of *ModelOperations*.

4. Specific-properties regarding *IO* operations within each submodel - see the *resolve* building-step, above.

3. Context-maps of `json_paths` –> *ModelOperations* instances, installed by *add_submodel()* and *unified_contexts* on the model-maker. They apply to self-or-descedant subtree of each model.

   The `json_path` is a strings obeying a simplified *json-pointer* syntax (no char-normalizations yet), ie `/some/foo/1/pointer`. An empty-string(`''`) matches all model.

   When multiple convertors match for a model-value, the selected convertor to be used is the most specific one (the one with longest prefix). For instance, on the model:

```
[ { "foo": { "bar": 0 } } ]
```

   all of the following would match the `0` value:

   - the global-default *_global_cntxt*,
   - `/`, and
   - `/0/foo`

   but only the last's context-props will be applied.

Atributes

`model`
The model-tree that will receive the merged submodels after *build()* has been invoked. Depending on the submodels, the top-value can be any of the supported model data-types.

_submodel_tuples
The stack of (`submodel`, `path_ops`) tuples. The list's 1st element is the *base-model*, the last one, the *top-model*. Use the *add_submodel()* to build this list.

_global_cntxt
A *ModelOperations* instance acting as the global-default context for the unified-model and all submodels. Use *mod_global_operations()* to modify it.

_curate_funcs
The sequence of *curate* functions to be executed as the final step by *_curate()*. They are "normal" functions (not generators) with signature:

```
def curate_func(model_maker):
    pass      ## ie: modify ``model_maker.model``.
```

Better specify this list of functions on construction time.

_errored
An internal boolean flag that becomes `True` if any build-step has failed, to halt proceeding to the next one. It is `None` if build has not started yet.

Examples

The basic usage requires to subclass your own model-maker, just so that a *json-schema* is provided for both validation-steps, 2 & 4:

```
>>> from collections import OrderedDict as od                    ## Json is better wit
```

```
>>> class MyModel(Pandel):
...     def _get_json_schema(self, is_prevalidation):
...         return {                                              ## Define the json-sc
...             '$schema': 'http://json-schema.org/draft-04/schema#',
...             'required': [] if is_prevalidation else ['a', 'b'],     ## Prevalidation is n
...             'properties': {
...                 'a': {'type': 'string'},
...                 'b': {'type': 'number'},
...                 'c': {'type': 'number'},
...             }
...         }
```

Then you can instanciate it and add your submodels:

```
>>> mm = MyModel()
>>> mm.add_submodel(od(a='foo', b=1))                            ## submodel-1 (base)
>>> mm.add_submodel(pd.Series(od(a='bar', c=2)))                 ## submodel-2 (top-mo
```

You then have to build the final unified-model (any validation errors would be reported at this point):

```
>>> mdl = mm.build()
```

Note that you can also access the unified-model in the *model* attribute. You can now interogate it:

```
>>> mdl['a'] == 'bar'                          ## Value overridden by top-model
True
>>> mdl['b'] == 1                              ## Value left intact from base-model
True
>>> mdl['c'] == 2                              ## New value from top-model
True
```

Lets try to build with invalid submodels:

```
>>> mm = MyModel()
>>> mm.add_submodel({'a': 1})                  ## According to the schema, this should have beer
>>> mm.add_submodel({'b': 'string'})           ## and this one, a number.
```

```
>>> sorted(mm.build_iter(), key=lambda ex: ex.message)                    ## Fetch a list with
[<ValidationError: "'string' is not of type 'number'">,
 <ValidationError: "1 is not of type 'string'">,
 <ValidationError: 'Gave-up building model after step 1.prevalidate (out of 4).'>]
```

```
>>> mdl = mm.model
>>> mdl is None                                          ## No model constructed, failed before me
True
```

And lets try to build with valid submodels but invalid merged-one:

```
>>> mm = MyModel()
>>> mm.add_submodel({'a': 'a str'})
>>> mm.add_submodel({'c': 1})
```

```
>>> sorted(mm.build_iter(), key=lambda ex: ex.message)          ## Missing required('b') prop r
[<ValidationError: "'b' is a required property">,
 <ValidationError: 'Gave-up building model after step 3.validate (out of 4).'>]
```

__init__ (*curate_funcs=()*)

> **Parameters** **curate_funcs** (*sequence*) – See *_curate_funcs*.

__metaclass__
alias of ABCMeta

_clone_and_merge_submodels (*a*, *b*, *path=u''*)
' Recursively merge b into a, cloning both.

_curate ()
Step-4: Invokes any curate-functions found in *_curate_funcs*.

_get_json_schema (*is_prevalidation*)

> **Returns** a json schema, more loose when `prevalidation` for each case
>
> **Return type** dictionary

_merge ()
Step-2

_prevalidate ()
Step-1

_read_branch ()
Reads model-branches during *resolve* step.

_resolve ()
Step-1

_select_context (*path*, *branch*)
Finds which context to use while visiting model-nodes, by enforcing the precedance-rules described in the *Customizations*.

> **Parameters**
>
> - **path** (*str*) – the branch's jsonpointer-path
> - **branch** (*str*) – the actual branch's node
>
> **Returns** the selected *ModelOperations*

_validate ()
Step-3

_write_branch ()
Writes model-branches during *distribute* step.

add_submodel(*model*, *path_ops=None*)
Pushes on top a submodel, along with its context-map.

Parameters

- **model** – the model-tree (sequence, mapping, pandas-types)
- **path_ops** (*dict*) – A map of `json_paths` –> *ModelOperations* instances acting on the unified-model. The `path_ops` may often be empty.

Examples

To change the default DataFrame –> dictionary convertor for a submodel, use the following:

```
>>> mdl = {'foo': 'bar'}
>>> submdl = ModelOperations(mdl, conv={(pd.DataFrame, dict): lambda df: df.to_dict('recc
```

build()
Attempts to build the model by exhausting *build_iter()*, or raises its 1st error.

Use this method when you do not want to waste time getting the full list of errors.

build_iter()
Iteratively build model, yielding any problems as `ValidationError` instances.

For debugging, the unified model at *model* my contain intermediate results at any time, even if construction has failed. Check the *_errored* flag if neccessary.

mod_global_operations(*operations=None*, ***cntxt_kwargs*)
Since it is the fall-back operation for *conversions* and *IO* operation, it must exist and have all its props well-defined for the class to work correctly.

Parameters

- **operations** (`ModelOperations`) – Replaces values of the installed context with non-empty values from this one.
- **cntxt_kwargs** – Replaces the keyworded-values on the existing `operations`. See *ModelOperations* for supported keywords.

unified_contexts
A map of `json_paths` –> *ModelOperations* instances acting on the unified-model.

class `wltp.pandel.`**PandelVisitor**(*schema*, *types=()*, *resolver=None*, *format_checker=None*, *skip_meta_validation=False*)
Bases: `jsonschema.validators.Validator`

A customized `Draft4Validator` suporting instance-trees with pandas and numpy objects, natively.

Any pandas or numpy instance (for example `obj`) is treated like that:

Python Type	JSON Equivalence
`pandas.DataFrame`	as `object` *json-type*, with `obj.columns` as *keys*, and `obj[col].values` as *values*
`pandas.Series`	as `object` *json-type*, with `obj.index` as *keys*, and `obj.values` as *values*
`np.ndarray`, `list`, `tuple`	as `array` *json-type*

Note that the value of each dataFrame column is a :`ndarray` instances.

The simplest validations of an object or a pandas-instance is like this:

```
>>> import pandas as pd
```

```
>>> schema = {
...     'type': 'object',
... }
>>> pv = PandelVisitor(schema)
```

```
>>> pv.validate({'foo': 'bar'})
>>> pv.validate(pd.Series({'foo': 1}))
>>> pv.validate([1,2])                                    ## A sequence is invalid here.
Traceback (most recent call last):
...
jsonschema.exceptions.ValidationError: [1, 2] is not of type 'object'

Failed validating 'type' in schema:
    {'type': 'object'}

On instance:
    [1, 2]
```

Or demanding specific properties with `required` and no `additionalProperties`:

```
>>> schema = {
...     'type':     'object',
...     'required': ['foo'],
...    'additionalProperties': False,
...    'properties': {
...        'foo': {}
...    }
... }
>>> pv = PandelVisitor(schema)
```

```
>>> pv.validate(pd.Series({'foo': 1}))
>>> pv.validate(pd.Series({'foo': 1, 'bar': 2}))          ## Additional 'bar' is present!
Traceback (most recent call last):
...
jsonschema.exceptions.ValidationError: Additional properties are not allowed ('bar' was unexp

Failed validating 'additionalProperties' in schema:
    {'additionalProperties': False,
     'properties': {'foo': {}},
     'required': ['foo'],
     'type': 'object'}

On instance:
    bar    2
    foo    1
    dtype: int64
```

```
>>> pv.validate(pd.Series({}))                            ## Required 'foo' missing!
Traceback (most recent call last):
...
jsonschema.exceptions.ValidationError: 'foo' is a required property

Failed validating 'required' in schema:
    {'additionalProperties': False,
     'properties': {'foo': {}},
     'required': ['foo'],
     'type': 'object'}

On instance:
    Series([], dtype: float64)
```

class `wltp.pandel.`**`PathMaps`**

Bases: `object`

Cascade prefix-mapping of json-paths to any values (here *`ModelOperations`*.

`wltp.pandel.`**`jsonpointer_parts`**(*jsonpointer*)

Iterates over the `jsonpointer` parts.

Parameters **jsonpointer** (*str*) – a jsonpointer to resolve within document

Returns a generator over the parts of the json-pointer

Author Julian Berman, ankostis

wltp.pandel.**resolve_jsonpointer**(*doc*, *jsonpointer*, *default=<object object>*)
Resolve a `jsonpointer` within the referenced `doc`.

Parameters

- **doc** – the referrant document
- **jsonpointer** (*str*) – a jsonpointer to resolve within document

Returns the resolved doc-item or raises `RefResolutionError`

Author Julian Berman, ankostis

wltp.pandel.**set_jsonpointer**(*doc*, *jsonpointer*, *value*, *object_factory=<type 'dict'>*)
Resolve a `jsonpointer` within the referenced `doc`.

Parameters

- **doc** – the referrant document
- **jsonpointer** (*str*) – a jsonpointer to the node to modify

Raises JsonPointerException (if jsonpointer empty, missing, invalid-contet)

6.4 Module: wltp.test.samples_db_tests

Compares the results of synthetic vehicles from JRC against pre-phase-1b Heinz's tool.

- Run as Test-case to generate results for sample-vehicles.
- Run it as cmd-line to compare with Heinz's results.

class wltp.test.samples_db_tests.**ExperimentSampleVehs**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Compares a batch of vehicles with results obtained from "Official" implementation.

test1_AvgRPMs()
Check mean-engine-speed diff with Heinz within some percent.

Results:

	mean	std	min	max
python	1876.555626	146.755857	1652.457262	2220.657166
heinz	1892.048584	148.248303	1660.710716	2223.772904
diff_prcnt	0.008256	0.010170	0.004995	0.001403

test1_PMRatio()
Check mean-engine-speed diff with Heinz within some percent for all PMRs.

Results:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(40.759, 49.936]	1814.752308	1822.011660	0.004000	4
(49.936, 59.00401]	1861.137208	1879.822876	0.010040	4
(59.00401, 68.072]	2015.693195	2031.240237	0.007713	3
(68.072, 77.14]	1848.735584	1859.116047	0.005615	5
(77.14, 86.208]	NaN	NaN	NaN	0
(86.208, 95.276]	1786.879366	1807.764020	0.011688	5
(95.276, 104.344]	1956.288657	1980.523043	0.012388	3
(104.344, 113.412]	1929.718933	1947.787155	0.009363	3

```
(113.412, 122.48]      2033.321183     2051.602998    0.008991      1
(122.48, 131.548]      1781.487338     1781.591893    0.000059      1
(131.548, 140.616]             NaN             NaN         NaN      0
(140.616, 149.684]     1895.125082     1907.872848    0.006727      1
```

wltp.test.samples_db_tests.**driver_weight = 70**
For calculating unladen_mass.

6.5 Module: wltp.test.wltp_db_tests

Compares the results of a batch of wltp_db vehicles against phase-1b-alpha Heinz's tool.

- Run as Test-case to generate results for sample-vehicles.
- Run it as cmd-line to compare with Heinz's results.

class wltp.test.wltp_db_tests.**WltpDbTests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Compares a batch of vehicles with results obtained from "official" implementation.

test1_Downscale()
Check mean-downscaled-velocity diff with Heinz within some percent.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
              python       heinz    diff_prcnt
count  378.000000  378.000000  0.000000e+00
mean    45.973545   46.189082  4.688300e-01
std      1.642335    1.126555 -4.578377e+01
min     35.866421   36.659117  2.210133e+00
25%     46.506718   46.504909 -3.892020e-03
50%     46.506718   46.506504 -4.620879e-04
75%     46.506718   46.506719  4.116024e-08
max     46.506718   46.506719  4.116024e-08
```

Not forcing class3b, honoring declared v_max & unladen_mass:

```
              python       heinz    diff_prcnt
count  382.000000  382.000000  0.000000e+00
mean    44.821337   44.846671  5.652189e-02
std      5.054214    5.050208 -7.933394e-02
min     28.091672   28.388418  1.056347e+00
25%     46.506718   46.504868 -3.978244e-03
50%     46.506718   46.506478 -5.162230e-04
75%     46.506718   46.506719  4.116033e-08
max     46.506718   46.506719  4.116033e-08
```

test2a_gear_diffs()
Check diff-gears with Heinz stays within some percent.

Comparison history

Class3b-Vehicles, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
              count       MEAN        STD  min  max
gears         23387  75.931818  56.921729    6  279
accell        19146  62.162338  48.831155    4  238
senza rules   16133  52.379870  35.858415   11  170
```

Separated test/unladen masses:

	diff_gears	diff_accel	diff_orig
count	378.000000	378.000000	378.000000
mean	104.965608	86.171958	90.235450
std	100.439783	82.613475	109.283901
min	6.000000	4.000000	11.000000
25%	36.250000	25.250000	23.000000
50%	69.000000	57.500000	51.000000
75%	142.000000	119.750000	104.750000
max	524.000000	404.000000	600.000000
sum	39677.000000	32573.000000	34109.000000
mean%	5.831423	4.787331	5.013081

Not forcing class3b, honoring declared v_max & unladen_mass:

	diff_gears	diff_accel	diff_orig
count	382.000000	382.000000	382.000000
mean	75.994764	63.633508	54.083770
std	58.290971	51.885162	38.762326
min	2.000000	2.000000	6.000000
25%	29.000000	22.000000	19.000000
50%	57.000000	48.500000	45.000000
75%	111.000000	97.000000	78.750000
max	279.000000	243.000000	173.000000
sum	29030.000000	24308.000000	20660.000000
mean%	4.221931	3.535195	3.004654

test2b_gear_diffs_transplanted()

Check driveability-only diff-gears with Heinz stays within some percent.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

	diff_gears	diff_accel	diff_orig
count	378.000000	378.000000	378
mean	15.566138	5.634921	0
std	16.554295	8.136700	0
min	0.000000	0.000000	0
25%	5.000000	1.000000	0
50%	11.000000	3.000000	0
75%	19.750000	7.000000	0
max	123.000000	78.000000	0
sum	5884.000000	2130.000000	0
mean%	0.864785	0.313051	0

Not forcing class3b, honoring declared v_max & unladen_mass:

	diff_gears	diff_accel	diff_orig
count	382.000000	382.000000	382
mean	12.599476	4.651832	0
std	15.375930	7.566103	0
min	0.000000	0.000000	0
25%	4.000000	0.000000	0
50%	9.000000	2.000000	0
75%	15.000000	6.000000	0
max	123.000000	78.000000	0
sum	4813.000000	1777.000000	0
mean%	0.699971	0.258435	0

test3a_n_mean()

Check mean-rpm diff with Heinz stays within some percent.

Comparison history

Class3b-Vehicles, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
                      mean         std          min          max
python      1766.707825  410.762478  1135.458463  3217.428423
heinz       1759.851498  397.343498  1185.905053  3171.826208
diff_prcnt    -0.3896     -3.3772       4.4428      -1.4377
```

Separated test/unladen masses:

```
             python        heinz  diff_prcnt
count   378.000000   378.000000    0.000000
mean   1923.908119  1899.366431   -1.292099
std     628.998854   593.126296   -6.048047
min    1135.458463  1185.905053    4.442839
25%    1497.544940  1495.699889   -0.123357
50%    1740.927971  1752.668517    0.674384
75%    2121.459309  2111.876041   -0.453780
max    4965.206982  4897.154914   -1.389625
```

Not forcing class3b, honoring declared v_max & unladen_mass:

```
             python        heinz  diff_prcnt
count   382.000000   382.000000    0.000000
mean   1835.393402  1827.572965   -0.427914
std     476.687485   464.264779   -2.675781
min    1135.458463  1185.905053    4.442839
25%    1486.886555  1482.789006   -0.276341
50%    1731.983662  1739.781233    0.450210
75%    2024.534101  2018.716963   -0.288160
max    3741.849187  3750.927263    0.242609
```

test3b_n_mean_transplanted()

Check driveability-only mean-rpm diff with Heinz stays within some percent.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
             python        heinz  diff_prcnt
count   378.000000   378.000000    0.000000
mean   1880.045112  1899.366431    1.027705
std     572.842493   593.126296    3.540904
min    1150.940393  1185.905053    3.037921
25%    1477.913404  1495.699889    1.203486
50%    1739.882957  1752.668517    0.734852
75%    2073.715015  2111.876041    1.840225
max    4647.136063  4897.154914    5.380063
```

Not forcing class3b, honoring declared v_max & unladen_mass:

```
             python        heinz  diff_prcnt
count   382.000000   382.000000    0.000000
mean   1818.519842  1827.572965    0.497829
std     469.276397   464.264779   -1.079474
min    1150.940393  1185.905053    3.037921
25%    1467.153958  1482.789006    1.065672
50%    1730.051632  1739.781233    0.562388
75%    2010.264758  2018.716963    0.420452
max    3704.999890  3750.927263    1.239605
```

test4a_n_mean__PMR()

Check mean-rpm diff with Heinz stays within some percent for all PMRs.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_ratio	count
(9.973, 24.823]	1566.018469	1568.360963	0.001496	32
(24.823, 39.496]	1701.176128	1702.739797	0.000919	32
(39.496, 54.17]	1731.541637	1724.959671	-0.003816	106
(54.17, 68.843]	1894.477475	1877.786294	-0.008889	61
(68.843, 83.517]	1828.518522	1818.720627	-0.005387	40
(83.517, 98.191]	1824.060716	1830.482140	0.003520	3
(98.191, 112.864]	1794.673461	1792.693611	-0.001104	31
(112.864, 127.538]	3217.428423	3171.826208	-0.014377	1
(127.538, 142.211]	1627.952896	1597.571904	-0.019017	1
(142.211, 156.885]	NaN	NaN	NaN	0
(156.885, 171.558]	NaN	NaN	NaN	0
(171.558, 186.232]	1396.061758	1385.176569	-0.007858	1

Separated test/unladen masses:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(11.504, 26.225]	1579.612698	1585.721306	0.386716	28
(26.225, 40.771]	1706.865069	1700.689983	-0.363093	41
(40.771, 55.317]	1866.150857	1841.779091	-1.323273	119
(55.317, 69.863]	2122.662626	2085.262950	-1.793523	122
(69.863, 84.409]	2228.282795	2171.952804	-2.593518	29
(84.409, 98.955]	1783.316413	1787.378401	0.227777	4
(98.955, 113.501]	1718.157828	1718.516147	0.020855	31
(113.501, 128.0475]	2005.415058	1954.763742	-2.591173	2
(128.0475, 142.594]	1566.601860	1553.383676	-0.850928	1
(142.594, 157.14]	NaN	NaN	NaN	0
(157.14, 171.686]	NaN	NaN	NaN	0
(171.686, 186.232]	1396.061758	1385.176569	-0.785834	1

Not forcing class3b, honoring declared v_max & unladen_mass:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(9.973, 24.823]	1560.010258	1563.836656	0.245280	33
(24.823, 39.496]	1725.209986	1725.004638	-0.011904	34
(39.496, 54.17]	1737.811065	1730.770088	-0.406812	123
(54.17, 68.843]	1996.999520	1983.753219	-0.667739	94
(68.843, 83.517]	2051.088434	2034.594136	-0.810692	59
(83.517, 98.191]	1964.832555	1958.081066	-0.344801	4
(98.191, 112.864]	1682.122484	1684.443875	0.138004	31
(112.864, 127.538]	2718.877009	2687.055802	-1.184241	2
(127.538, 142.211]	1660.925042	1668.155469	0.435325	1
(142.211, 156.885]	NaN	NaN	NaN	0
(156.885, 171.558]	NaN	NaN	NaN	0
(171.558, 186.232]	1396.061758	1385.176569	-0.785834	1

Mean: 0.419219429398

pandas 0.15.1:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(9.973, 24.823]	2037.027221	2038.842442	0.089111	33
(24.823, 39.496]	2257.302959	2229.999526	-1.224369	34
(39.496, 54.17]	1912.075914	1885.792807	-1.393743	123
(54.17, 68.843]	1716.720028	1717.808457	0.063402	94
(68.843, 83.517]	1677.882399	1683.916224	0.359610	59
(83.517, 98.191]	1535.881170	1551.609661	1.024070	4
(98.191, 112.864]	1571.290286	1589.997331	1.190553	31
(112.864, 127.538]	1409.308426	1425.965019	1.181898	2
(127.538, 142.211]	1975.481368	1967.808440	-0.389923	1
(142.211, 156.885]	NaN	NaN	NaN	0

```
(156.885, 171.558]             NaN             NaN         NaN      0
(171.558, 186.232]     1950.377512     1937.426430   -0.668468      1
Mean diff_prcnt: 0.632095580562
```

test4b_n_mean__PMR_transplanted()

Check driveability-only mean-rpm diff with Heinz stays within some percent for all PMRs.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(9.973, 24.823]	1557.225037	1568.360963	0.715113	32
(24.823, 39.496]	1686.859826	1696.482640	0.570457	34
(39.496, 54.17]	1771.670097	1789.409819	1.001299	120
(54.17, 68.843]	2133.400050	2165.214662	1.491263	94
(68.843, 83.517]	2020.903728	2043.741660	1.130085	59
(83.517, 98.191]	1886.836446	1890.040533	0.169813	4
(98.191, 112.864]	1788.434592	1792.693611	0.238142	31
(112.864, 127.538]	2580.884314	2568.011660	-0.501269	2
(127.538, 142.211]	1581.625191	1597.571904	1.008249	1
(142.211, 156.885]	NaN	NaN	NaN	0
(156.885, 171.558]	NaN	NaN	NaN	0
(171.558, 186.232]	1367.068837	1385.176569	1.324566	1

Separated test/unladen masses:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(11.504, 26.225]	1572.733597	1585.721306	0.825805	28
(26.225, 40.771]	1690.081663	1700.689983	0.627681	41
(40.771, 55.317]	1821.319706	1841.779091	1.123327	119
(55.317, 69.863]	2060.507029	2085.262950	1.201448	122
(69.863, 84.409]	2142.964427	2171.952804	1.352723	29
(84.409, 98.955]	1783.214173	1787.378401	0.233524	4
(98.955, 113.501]	1713.473617	1718.516147	0.294287	31
(113.501, 128.0475]	1950.373771	1954.763742	0.225084	2
(128.0475, 142.594]	1543.937285	1553.383676	0.611838	1
(142.594, 157.14]	NaN	NaN	NaN	0
(157.14, 171.686]	NaN	NaN	NaN	0
(171.686, 186.232]	1367.068837	1385.176569	1.324566	1

Not forcing class3b, honoring declared v_max & unladen_mass:

pmr	gened_mean_rpm	heinz_mean_rpm	diff_prcnt	count
(9.973, 24.823]	1551.901645	1563.836656	0.769057	33
(24.823, 39.496]	1713.382835	1725.004638	0.678296	34
(39.496, 54.17]	1722.174466	1730.770088	0.499114	123
(54.17, 68.843]	1974.768859	1983.753219	0.454958	94
(68.843, 83.517]	2026.630271	2034.594136	0.392961	59
(83.517, 98.191]	1954.817179	1958.081066	0.166966	4
(98.191, 112.864]	1676.678357	1684.443875	0.463149	31
(112.864, 127.538]	2678.973439	2687.055802	0.301696	2
(127.538, 142.211]	1658.577318	1668.155469	0.577492	1
(142.211, 156.885]	NaN	NaN	NaN	0
(156.885, 171.558]	NaN	NaN	NaN	0
(171.558, 186.232]	1367.068837	1385.176569	1.324566	1

Mean diff_prcnt: 0.469021296461

pandas 0.15.1:

```
                      gened_mean_rpm  heinz_mean_rpm  diff_prcnt  count
pmr
(9.973, 24.823]         2021.882193     2038.842442    0.838835     33
(24.823, 39.496]        2204.136804     2229.999526    1.173372     34
(39.496, 54.17]         1880.733341     1885.792807    0.269016    123
(54.17, 68.843]         1710.819917     1717.808457    0.408491     94
(68.843, 83.517]        1677.846860     1683.916224    0.361735     59
(83.517, 98.191]        1541.587174     1551.609661    0.650141      4
(98.191, 112.864]       1579.049392     1589.997331    0.693325     31
(112.864, 127.538]      1411.921405     1425.965019    0.994646      2
(127.538, 142.211]      1976.193317     1967.808440   -0.426102      1
(142.211, 156.885]              NaN             NaN         NaN      0
(156.885, 171.558]              NaN             NaN         NaN      0
(171.558, 186.232]      1954.662077     1937.426430   -0.889616      1
Mean diff_prcnt: 0.558773102894
```

test5a_n_mean__gear()

Check mean-rpm diff% with Heinz stays within some percent for all gears.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
n_mean       python        heinz       diff%
gear
0        732.358286   804.656085  -9.925769
1        870.080494  1177.547512 -44.450903
2       1789.787609  1650.383967   6.520319
3       1921.271483  1761.172027   7.804359
4       1990.286402  1886.563262   5.401895
5       2138.445024  2112.552162   1.892950
6       2030.970322  1987.865039   2.228276
```

Not forcing class3b, honoring declared v_max & unladen_mass:

```
gear
0         735.143823   808.795812 -10.052865
1         799.834530  1139.979330 -47.027383
2        1598.773915  1582.431975   1.119054
3        1793.617644  1691.589756   5.768020
4        1883.863510  1796.957457   5.024360
5        2095.211754  2052.059948   2.430360
6        2033.663975  1990.344346   2.238421
```

test5b_n_mean__gear_transplanted()

Check mean-rpm diff% with Heinz stays within some percent for all gears.

Comparison history

Force class3b, Phase-1b-beta(ver <= 0.0.8, Aug-2014) with Heinz maxt gear-time=2sec:

```
n_mean       python        heinz       diff%
gear
0        732.357001   804.656085  -9.926855
1        966.022039  1177.547512 -24.409425
2       1678.578373  1650.383967   1.616768
3       1791.644768  1761.172027   1.700642
4       1883.504933  1886.563262   0.119165
5       2099.218160  2112.552162  -0.320293
6       1985.732086  1987.865039  -0.096754
```

Not forcing class3b, honoring declared v_max & unladen_mass:

```
n_mean       python        heinz       diff%
gear
0         735.077116   808.795812 -10.065886
```

```
1        932.586982  1139.979330 -24.285307
2       1606.040896  1582.431975   1.379144
3       1721.141364  1691.589756   1.686708
4       1803.212699  1796.957457   0.370703
5       2053.822313  2052.059948   0.142138
6       1988.195381  1990.344346  -0.097482
```

wltp.test.wltp_db_tests.**_file_pairs**(*fname_glob*)

Generates pairs of files to compare, skipping non-existent and those with mismatching #_of_rows.

Example:

```
>>> for (veh_num, df_g, df_h) in _file_pairs('wltp_db_vehicles-00*.csv')
        pass
```

wltp.test.wltp_db_tests.**aggregate_single_columns_means**(*gened_column*, *heinz_column*)

Runs experiments and aggregates mean-values from one column of each (gened, heinz) file-sets.

wltp.test.wltp_db_tests.**driver_weight = 70**

For calculating unladen_mass.

wltp.test.wltp_db_tests.**vehicles_applicator**(*fname_glob*, *pair_func*)

Applies the fun onto a pair of (generated, heinz) files for each tested-vehicle in the glob and appends results to list, preffixed by veh_num.

Parameters **pair_func** – signature: func(veh_no, gened_df, heinz_df)–>sequence_of_numbers

Returns a dataframe with the columns returned from the pair_func, row_indexed by veh_num

CHAPTER 7

Changes

Contents

7.1 GTR version matrix

Given a version number `MAJOR.MINOR.PATCH`, the `MAJOR` part tracks the GTR phase implemented. The following matrix shows these correspondences:

Release train	GTR ver
0.x.x	Till Aug 2014, Not very Precise with the till-that-day standard. (diffs explained below)
1.x.x	After Nov 2014, phase 2b (TBD)

7.2 Known deficiencies

- (!) Driveability-rules not ordered as defined in the latest task-force meeting.
- (!) The driveability-rules when speeding down to a halt is broken, and human-drivers should improvise.
- (!) The `n_min_drive` is not calculated as defined in the latest task-force meeting, along with other recent updates.

- (!) The `n_max` is calculated for ALL GEARS, resulting in "clipped" velocity-profiles, leading to reduced rpm's for low-powered vehicles.
- Clutching-points and therefore engine-speed are very preliminary (ie `rpm` when starting from stop might be < `n_idle`).

7.3 TODOs

- Add cmd-line front-end.
- Automatically calculate masses from H & L vehicles, and regression-curves from categories.
- wltp_db: Improve test-metrics with group-by classes/phases.
- model: Enhance model-preprocessing by interleaving "octapus" merging stacked-models between validation stages.
- model: finalize data-schema (renaming columns and adding `name` fields in major blocks).
- model/core: Accept units on all quantities.
- core: Move calculations as class-methods to provide for overriding certain parts of the algorithm.
- core: Support to provide and override arbitrary model-data, and ask for arbitrary output-ones by topologically sorting the graphs of the calculation-dependencies.
- build: Separate wltpdb tests as a separate, optional, plugin of this project (~650Mb size).

7.4 Releases

7.4.1 v0.0.9-alpha.1, alpha.3 (1 Oct, X Noe 2014)

This is practically the 2nd public releases, reworked in many parts, and much better documented and continuously tested and build using TravisCI, BUT the arithmetic results produced are still identical to v0.0.7, so that the test-cases and metrics still describe this core.

Important/*incompatilble* changes

- **Code:**
  - package `wltc` –> `wltp`
  - class `Experiment` –> `Processor`
- **Model changes:**
  - `/vehicle/mass` –> (`test_mass` and `unladen_mass`)
  - `/cycle_run`: If present, (some of) its columns override the calculation.
- Added tkUI and Excel front-ends.

Changelog

v0.0.9-alpha.3

Shared with LAT. * Use CONDA for running no TravisCI. * Improve ExcelRunner. * docs and metrics improvments.

v0.0.9-alpha.2

- ui: Added Excel frontend.
- ui: Added desktop-UI proof-of-concept (`wltp.tkui`).
- metrics: Add diagrams auto-generated from test-metrics into generated site (at "Getting Involved" section).

v0.0.9-alpha.1

- Backported also to Python-2.7.
- model, core: Discriminate between *Test mass* from *Unladen mass* (optionally auto-calced by `driver_mass` = 75(kg)).
- model, core: Calculate default resistance-coefficients from a regression-curve (the one found in Heinz-db).
- model, core: Possible to overide WLTP-Class, Target-V & Slope, Gears if present in the `cycle_run` table.
- model: Add NEDC cycle data, for facilitating comparisons.
- tests: Include sample-vehicles along with the distribution.
- tests: Speed-up tests by caching files to read and compare.
- docs: Considerable improvements, validate code in comments and docs with *doctest*.
- docs: Provide a http-link to the list of IPython front-ends in the project's wiki.
- build: Use TravisCI as integration server, Coveralls.io as test-coverage service-providers.
- build: Not possible anymore to distribute it as .EXE; need a proper python-3 environment.

7.4.2 v0.0.8-alpha, 04-Aug-2014

- Documentation fixes.

7.4.3 v0.0.7-alpha, 31-Jul-2014: 1st *public*

Although it has already been used in various exercises, never made it out of *Alpha* state.

- Rename project to 'wltp'.
- Switch license from AGPL –> EUPL (the same license assumed *retrospectively* for older version)
- Add wltp_db files.
- Unify instances & schemas in `model.py`.
- Possible to Build as standalone `exe` using `cx_freeze`.
- **Preparations for PyPI/github distribution.**
  - Rename project to "wltp".
  - Prepare Sphinx documentation for http://readthedocs.org.
  - Update setup.py
  - Update project-coordinates (authors, etc)

7.4.4 v0.0.6-alpha, 5-Feb-2014

- Make it build as standalone `exe` using `cx_freeze`.
- Possible to transplant base-gears and then apply on them driveability-rules.
- Embed Model –> Experiment to simplify client-code.
- Changes in the data-schema for facilitating conditional runs.
- More reverse-engineered comparisons with heinz's data.

7.4.5 v0.0.5-alpha, 18-Feb-2014

- Many driveability-improvements found by trial-n-error comparing with Heinz's.
- Changes in the data-schema for facilitating storing of tabular-data.
- Use Euro6 polynomial full_load_curve from Fontaras.
- Smooth-away INALID-GEARS.
- Make the plottings of comparisons of sample-vehicle with Heinz'results interactively report driveability-rules.
- Also report GEARS_ORIG, RPM_NORM, P_AVAIL, RPM, GEARS_ORIG, RPM_NORM results.

7.4.6 v0.0.4.alpha, 18-Jan-2014

- Starting to compare with Heinz's data - FOUND DISCREPANCIES IMPLTYING ERROR IN BASE CALCS.
- Test-enhancements and code for comparing with older runs to track algo behavior.
- Calc 'V_real'.
- Also report RPMS, P_REQ, DIRVEABILITY results.
- Make v_max optionally calculated from max_gear / gear_ratios.
- BUGFIX: in P_AVAIL 100% percents were mixed [0, 1] ratios!
- BUGFIX: make `goodVehicle` a function to avoid mutation side-effects.
- BUGFIX: add forgotten division on p_required Accel/3.6.
- BUGFIX: velocity-profile mistakenly rounded to integers!
- BUGFIX: v_max calculation based on n_rated (not 1.2 * n_rated).
- FIXME: get default_load_curve floats from Heinz-db.
- FIXME: what to to with INVALID-GEARS?

7.4.7 v0.0.3_alpha, 22-Jan-2014

- **-Driveability rules not-implemented:**
  - missing some conditions for rule-f.
  - no test-cases.
  - No velocity_real.
  - No preparation calculations (eg. vehicle test-mass).
  - Still unchecked for correctness of results.

- **-Pending Experiment tasks:**
  - FIXME: Apply rule(e) also for any initial/final gear (not just for i-1).
  - FIXME: move V–0 into own gear.
  - FIXME: move V–0 into own gear.
  - FIXME: NOVATIVE rule: "Clutching gear-2 only when Decelerating.".
  - FIXME: What to do if no gear foudn for the combination of Power/Revs??
  - NOTE: "interpratation" of specs for Gear-2
  - NOTE: Rule(A) not needed inside x2 loop.
  - NOTE: rule(b2): Applying it only on non-flats may leave gear for less than 3sec!
  - NOTE: Rule(c) should be the last rule to run, outside x2 loop.
  - NOTE: Rule(f): What if extra conditions unsatisfied? Allow shifting for 1 sec only??
  - TODO: Construct a matrix of n_min_drive for all gears, including exceptions for gears 1 & 2.
  - TODO: Prepend row for idle-gear in N_GEARS
  - TODO: Rule(f) implement further constraints.
  - TODO: Simplify V_real calc by avoiding multiply all.

7.4.8 v0.0.2_alpha, 7-Jan-2014

- -Still unchecked for correctness of results.

7.4.9 v0.0.1, 6-Jan-2014: Alpha release

- -Unchecked for correctness.
- Runs OK.
- Project with python-packages and test-cases.
- Tidied code.
- Selects appropriate classes.
- Detects and applies downscale.
- Interpreted and implemented the nonsensical specs concerning `n_min` engine-revolutions for gear-2 (Annex 2-3.2, p71).
- -Not implemented yet driveability rules.
- -Does not output real_velocity yet - inly gears.

7.4.10 v0.0.0, 11-Dec-2013: Inception stage

- Mostly setup.py work, README and help.

CHAPTER 8

Indices

8.1 Glossary

WLTP The Worldwide harmonised Light duty vehicles Test Procedure, a *GRPE* informal working group

UNECE The United Nations Economic Commission for Europe, which has assumed the steering role on the *WLTP*.

GRPE *UNECE* Working party on Pollution and Energy - Transport Programme

GS Task-Force The Gear-shift Task-force of the *GRPE*. It is the team of automotive experts drafting the gear-shifting strategy for vehicles running the *WLTP* cycles.

WLTC The family of pre-defined *driving-cycles* corresponding to vehicles with different PMR (Power to Mass Ratio). Classes 1,2, 3a & 3b are split in 2, 4, 4 and 4 *parts* respectively.

Unladen mass *UM* or *Curb weight*, the weight of the vehicle in running order minus the mass of the driver.

Test mass *TM*, the representative weight of the vehicle used as input for the calculations of the simulation, derived by interpolating between high and low values for the CO_2-family of the vehicle.

Downscaling Reduction of the top-velocity of the original drive trace to be followed, to ensure that the vehicle is not driven in an unduly high proportion of "full throttle".

pandas-model The *container* of data that the gear-shift calculator consumes and produces. It is implemented by `wltp.pandel.Pandel` as a mergeable stack of *JSON-schema* abiding trees of strings and numbers, formed with sequences, dictionaries, `pandas`-instances and URI-references.

JSON-schema The JSON schema is an IETF draft that provides a *contract* for what JSON-data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data. You can learn more about it from this excellent guide, and experiment with this on-line validator.

JSON-pointer JSON Pointer(**RFC 6901**) defines a string syntax for identifying a specific value within a JavaScript Object Notation (JSON) document. It aims to serve the same purpose as *XPath* from the XML world, but it is much simpler.

8.1.1 Index

CHAPTER 9

Glossary

WLTP The Worldwide harmonised Light duty vehicles Test Procedure, a *GRPE* informal working group

UNECE The United Nations Economic Commission for Europe, which has assumed the steering role on the *WLTP*.

GRPE *UNECE* Working party on Pollution and Energy - Transport Programme

GS Task-Force The Gear-shift Task-force of the *GRPE*. It is the team of automotive experts drafting the gear-shifting strategy for vehicles running the *WLTP* cycles.

WLTC The family of pre-defined *driving-cycles* corresponding to vehicles with different PMR. Classes 1,2, 3a & 3b are split in 2, 4, 4 and 4 *parts* respectively.

Unladen mass *UM* or *Curb weight*, the weight of the vehicle in running order minus the mass of the driver.

Test mass *TM*, the representative weight of the vehicle used as input for the calculations of the simulation, derived by interpolating between high and low values for the CO_2-family of the vehicle.

Downscaling Reduction of the top-velocity of the original drive trace to be followed, to ensure that the vehicle is not driven in an unduly high proportion of "full throttle".

pandas-model The *container* of data that the gear-shift calculator consumes and produces. It is implemented by `wltp.pandel.Pandel` as a mergeable stack of *JSON-schema* abiding trees of strings and numbers, formed with sequences, dictionaries, `pandas`-instances and URI-references.

JSON-schema The JSON schema is an IETF draft that provides a *contract* for what JSON-data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data. You can learn more about it from this excellent guide, and experiment with this on-line validator.

JSON-pointer JSON Pointer(**RFC 6901**) defines a string syntax for identifying a specific value within a JavaScript Object Notation (JSON) document. It aims to serve the same purpose as *XPath* from the XML world, but it is much simpler.

W

Symbols

A

B

C

D

E

G

J

M

P

R

S

T

U

V

W