# NetGen

The following consists of a brief presentation of NetGen as a software and a user manual for people interested in using this tool for generating consistent datasets amenable for analysis in a few steps.

## *Technical Foreword*

NetGen is coded in Python. The current version does not support any graphical user interface (GUI) in order to ensure platform independence, that is, NetGen runs on any operating systems (Windows, OS X, Linux), any version of Python and with any user account limitations (as it does not use any extra library). In order to run it, a Python environment needs to be installed on the computer. Nevertheless, NetGen does not depend on the environment used (IDLE, Eclipse, Terminal, etc). Inside the Bundesbank, the only permitted environment is the Python IDLE. There exist many versions (e.g. 2.4, 2.7, 3.3) but any will work with NetGen. Note that no prior knowledge of Python will be needed to run the program.

Even though no programming skills are required, we here provide a list of good habits to maintain when dealing with scripts.

- Avoid accents
- Use different names for different variables
- Avoid spaces
- Avoid using files not present in the main NetGen folder or any sub folder

The data description of the German interbank market presents many challenges in cleaning and preparing datasets. In the following, we refer to the general set of extra-actions that are needed in such context as *mapping processes*. Hence, a mapping process can refer to a challenge such as identifiers matching, dropping of certain entries, mergings, etc.

## *A First Look at NetGen*

NetGen comes as a zip file that can be stored and extracted anywhere in the user's drive or on a remote server. The directory contains subfolders and two Python files (Figure 1).

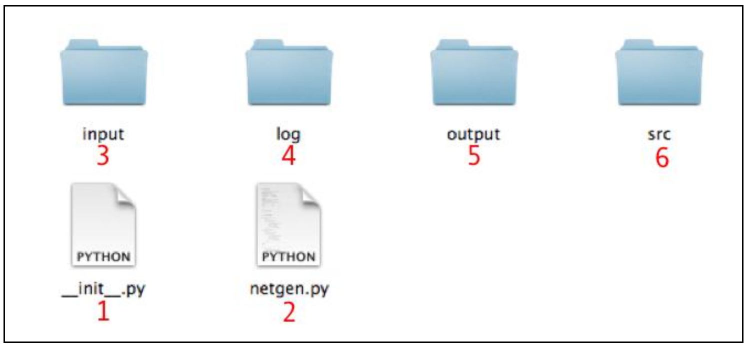

Figure 1: The NetGen Folder

1. The `__init__.py` Python file is required for Python to treat the different packages. It is a mandatory file for NetGen to work properly. As no interaction is needed with it, the user can ignore this element.
2. The `netgen.py` Python file is the main script of the program. This means that it is the file that needs to be executed via the IDLE in order to launch NetGen. It automatically uses the other modules and adapts to the configuration provided by the user (see next section).
3. The `input/` folder is the (default) directory in which all the files needed as input are to be stored: raw data file(s) and any additional mapping file(s).
4. The `output/` folder is the default directory where all the files produced by NetGen will be stored.[1]
5. The `log/` folder contains logs of previous NetGen runs. Those logs retrieve information on the amount of data that is being dropped or merged during the different mapping processes. It is helpful for tracking the impact of the configuration being implemented.
6. The `src/` folder contains all the modules used by NetGen (Figure 2).

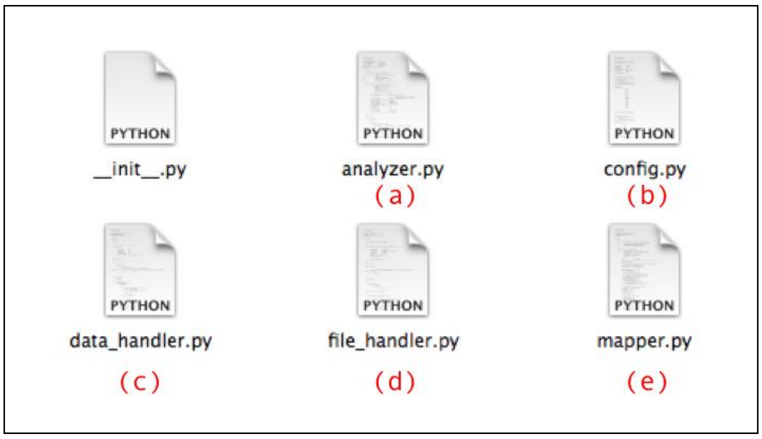

Figure 2: The `src/` folder

(a) The **Analyzer** (`analyzer.py`) is the general module called by NetGen to dispatch any task among the other more specialized modules. It provides the interface and the translation between the specific tasks and the end-user instructions.
(b) The **Configuration** file (`config.py`) contains all the information that the user needs to provide for NetGen to run. It is not a script per se but rather a collection of configuration information: the input file(s), output file(s), the data structure (raw and cleaned) and the mapping processes.
(c) The **Data Handler** (`data_hanlder.py`) is the module specialized in dealing with data. It verifies consistency, checks for outliers and inconsistent entries and structures them according to the configuration entered by the user.
(d) The **File Handler** (`file_handler.py`) is the module specialized in dealing with external files (raw data files, output files, and mapping files). It allows importing data, storing results and dealing with missing values.
(e) The **Mapper** (`mapper.py`) is the module responsible for taking all extra-action into account and applying them to the cleaned data. Among others, it allows to match

1 Both directories (input and output) are not mandatory. The user can use different directories by defining their absolute path during the configuration process (see next section).

identifiers, consolidate many identifiers under one identifier, drop entries not referenced in the mapping files, merge entries, etc.

## *How to Run NetGen*

Three steps are needed before running NetGen.

1. Input files preparation:

The user assembles all needed files (raw data files and mapping files), identifies their location, formats, and inner structures (delimiter, etc.).

2. Configuration of the `config` file:

The user fills *all* the entries in in the file `config.py`. This is the most important step throughout the use of NetGen. It will be described in a detailed manner in the following section.

3. Launching NetGen:

This step merely consists of opening the file `netgen.py` via the IDLE and running it by pressing the F5-key.

## *The Configuration Process*

This section further describes the configuration process presented in the second step of the previous section. It consists of a simple script that the user needs to fill. In the script, lines starting with a '#' are comments placed to give guidance to the user. All lines not starting with '#' are entries that need the user's input.

The user reports settings in single quotes after the equal symbol as follows[2]:

```
configuration_entry = 'user_entry'
```

Note the following important rule: all entries must be filled; if no specific values is to be entered by the user for one entry, this entry must be equal to an empty single quote (`''`). The rest of this section is divided in four parts. Each part describes a set of entries present in the `config` file. Their order follows the ordering in which they appear in the `config` script.

### 1. File description

Each file that will be used by NetGen (as input or output) must be introduced using the following information: directory, name, extension and delimiter.
The `directory` indicates where the file is located (will be written) for input files (output files) and should always end with '/'. If the directory is a sub folder of the NetGen folder, the relative path can be given, that is, a path starting from the NetGen directory. If not, the user has to introduce an absolute path, that is, a path starting from the main drive directory. The `name` is the name of the file: everything before the point. The `extension` is the format of the file: the three letters after the point. The `delimiter` indicates how elements are separated in each line of the file.

2 The last part of this section will describe how the user can deal with several values for one entry, that is, dealing with multiple files and multiple processes.

Here is an example of a text file located in the `input/` sub folder named ’`agenda`’ in which elements are separated via comas.

```
file_directory = 'input/'
file_name = 'agenda'
file_extension = 'txt'
file_delimiter = ','
```

Additionally, the input and output file description also allows defining how missing values are reported. The input data files can also be defined as being quoted or not.

Finally, when several raw data files are given, the user can also decide if all results should be stored into one or several files. The latter case will be described in the last part of this section. For the former case, the entry `out_file_single_file` must be activated (`'on')`. The user can also decide how results from different input files should be stored by filling the `out_file_separate_line` entry. There are three possibilities:

- ’`anytext`’ : the user entries a text that will always appear between the results of different datafiles
- ’`in_file_name`’ : by entering ’`in_file_name`’ the user specifies that the separating line should contain as text the name of the raw data file from which the coming results are produced
- ’’ : by leaving the entry empty, the use indicates that the resulting unique data file will not provide any separation between results coming from different raw data files.

Here is an example where the user activates the single output file option and specifies that results coming from different datasets must be separated by a line containing ’`New set of results:`’

```
out_file_single_file = 'on'
out_file_separate_line = 'New set of results:'
```

## 2. Data structure and cleaning process

For each input raw datafile, the user needs to describe the inner structure of the data, that is, how many fields exist per line and what format should they be associated with. This is achieved by completing the ’`raw_data_structure`’ entry in the `config` file. The protocol is to open a list by opening brackets (`[]`) where the user gives for each field a name and the format of the values. Each input must be quoted and separated via comas. Here is an example:

```
raw_data_structure = [ 'From', 'str', 'To', 'str', 'Exposure',
                      'float', 'IRI1','str']
```

The following formats are currently admitted:

- `'str'`= strings - any text
- `'float'`= floats - any real number
- `'int'`= integer - any integer
- `'date'`= date in the format day/month/year

Following the same protocol, the user then describes the structure expected for the output data in the '`clean_data_structure`'. Names reported in this entry must refer to names previously reported in the raw data structure. To follow with the previous example, here is one possible clean data structure:

```
clean_data_structure = [ 'From', 'str', 'To', 'str', 'Exposure',
                                'float']
```

The resulting data would then only consist of entries made of `From`, `To` and `Exposure`.

## 3. Mapping process

The mapping procedure consists of a set of controls that allows tackling all challenges reported in the previous section describing the data. All mapping processes are based upon additional information that needs to be retrieved through additional files which will also need to be defined in the `config` file (directory, name, extension, delimiter).

- Replacing values

This process allows changing values present in the data given a mapping list from a given mapping file.

Example of use: this is typically what will be used to match Geber and Nehmer identifiers.

To run this process, the user needs to (i) activate the related control, (ii) indicate which values to look for in the data, (iii) what to replace them with and (iv) which fields in the data should be checked during the search.

The points (ii) and (iii) define positions in the mapping file. Here is the example of an entry in a mapping file matching the lender and borrower identifier of a bank.

```
0002,1002
```

The position of the lender identifier is the column 0 and the position of the borrower identifier is the column 1. Thus, the related entries in the `config` file should be as follows:

```
mapping_replace_ids = 'on'
mapping_kept_id_position = '0'
mapping_lost_id_position = '1'
mapping_target_position = 'To'
```

This example replaces borrower identifies with lender identifiers. Additionally, it specifies the target position where the change should exclusively be performed: in the column '`To`' of the cleaned data. If the target position field is left empty ('') or with the value '`all`' then all the fields of the entry will be checked for replacement. If several columns are to be checked, but not all of them, columns' names must be indicated between the single quote separated by coma (ex: '`From,To`')

- Dropping unreferenced entities

This process allows the user to force the dropping of any entry that does not contain certain values present in the list of kept identifiers in the mapping file.

Example of use: this is typically what will be used to drop entries related to firms in the Exposure data in order to make sure that the resulting dataset only contains banks.

To run this process, the user needs to (i) activate the control and (ii) define targets specifying which entries in the data should be checked. Here is an example where the mapping process will drop any entry which does not have in the '`From`' field and in the '`To`' field, an identifier not present in the kept identifiers of the mapping file:

```
mapping_drop_unreferenced_entries = 'on'
mapping_target_unreferenced_entries = 'From,To'
```

The same protocol as for the matching values control stands for the target entry.

- <u>Removing duplicates</u>

This process allows the user to make sure no double counting is present in the data.

Example of use: this is typically what will be used to remove duplicates present when sub entities are lending to the same counterparty as a reporting artifact from their holding institution.

To run this process, the user needs to (i) activate the control and (ii) define targets specifying the set containing the fields in the data that will be compared in order to determine duplication. Here is an example where the process will consider duplicates by comparing one by one all fields of different entries and subsequently drop one.

```
mapping_remove_duplicates = 'on'
mapping_target_duplicates_set = 'all'
```

The same protocol as for the matching values control stands for the target entry.

- <u>Merging lines</u>

This process allows the user to merge entries.

Example of use: this is typically what will be used after entities have been matched to their holding institution identifier allowing for several entries to exist between to same institutions.

To run this process, the user needs to (i) activate the control, (ii) define targets specifying the set containing the fields in the data that will be compared to determine whether two entries must be merged and (iii) the set of commands to apply to the rest of fields in the entry. Here is an example:

```
mapping_merge_entries = 'on'
mapping_target_merge_set = 'From,To'
mapping_commands = 'Exposure : avg'
```

The same protocol as for the matching values control stands for the target entry. The protocol for the commands is the following: for each field not present in the target set, the user specifies which action to take between the corresponding values in the different entries in the following way:

```
'field_name1:action1, field_name2:action2'
```

That is, each instruction is separated by a coma inside the single quotation, as it was done for specifying the targets. Three commands can be performed:

- `'+'`: aggregation – summing up values
- `'avg'`: taking the average values of all the merged entries
- `'same'`: keeping the value if it is the same in all entries, indicating a missing value otherwise.

If existing fields are not explicitly mentioned in the command, the default action is '`same`'.

- Minor: Dropping ghosts.

This control allows specifying that if a variable in the mapping file has an empty field in the kept position, all entries containing that field should be dropped. The scope of this removal is implicitly related to the target positions defined by the value of the entry `mapping_kept_id_position`.

## 4. Multiple files and processes

For any step where input files are needed, it is possible to use several files (e.g. several raw data files, several steps for the full mapping process using different configuration including several mapping files). To do this, the user can use brackets and comas to separate the different files, that is, make a list of instructions. For example, the previous input file example becomes:

```
file_directory   = ['input/period1/' , 'input/period2/']
file_name        = ['agenda1','agenda2']
file_extension   = ['txt','txt']
file_delimiter   = [',',',']
```

This case presents two files to be used as input: one present in the subfolder '`period1/`' and the other present in the subfolder '`period2/`'. The sequence of information related to the same file in the different entries of the `config` file must be respected as it is shown in the example: file '`agenda1`' is considered to exist in the sub folder '`period1/`' and '`agenda2`' in '`period2/`'.

As it can often be the case, some entries might be the same for all files. In this case, the user does not need to repeat the information. He can simply provide one information and NetGen will infer that this information stands for all cases. In the above example, both files have the same extension and the same delimiter. The extension and the delimiter can thus be modified in the following way:

```
file_directory = ['input/period1/' , 'input/period2/']
file_name = ['agenda1','agenda2']
file_extension = 'txt'
file_delimiter = ','
```

Nevertheless, as soon as one file needs different information for one entry, the full sequence must then explicitly be reported.