A flexible data generator for privacy-preserving data mining and record linkage

Release 0.1

Peter Christen and Dinusha Vatsalan

March 30, 2012

Research School of Computer Science
ANU College of Engineering and Computer Science
The Australian National University
Canberra ACT 0200
Australia
Email: peter.christen@anu.edu.au, dinusha.vatsalan@anu.edu.au

Abstract

This manual describes prototype software for the flexible generation of data sets that contain randomly created personal information that is based on real data. Each of the records in such a data set is created individually. The generator allows the specification of different attribute types, including attributes based on frequency look-up table, functions, as well as several types of compound attributes where the values in one attribute depend upon the values in one or two other attributes.

The manual describes the structure of the generator, its program modules and the classes and methods within these modules, how to use and customise the generator, the format of look-up tables used, how the generator can be installed, and how its functionality can be tested.

The generator software is licensed under the **Mozilla Public License Version 2**, which is included in Appendix A.

CONTENTS

CHAPTER

ONE

INTRODUCTION

Research and development of novel techniques in areas such as data mining and record linkage commonly require real-world data for testing and evaluation of the accuracy, efficiency and effectiveness of these techniques. In areas and applications where the data include personal information, it is often difficult to acquire real data due to privacy and confidentiality concerns [2].

An alternative to using real data is to generate synthetic (or artificial) data which is based on real data [1, 5]. Such generated data should exhibit similar statistical characteristics compared to the real data they are based on. For example, the generated values, their frequency distributions, and the occurrences and frequencies of typographical and other errors and variations should follow real data. Dependencies between elements of real data should also be modelled.

The methodological advantages of synthetic data are that [1, 5] (a) they can be generated with well defined frequency distributions, error characteristics and variations; (b) the status of which records have been generated based on each other is known, allowing for example to measure the accuracy of matching records when synthetic data are used for record linkage (generally not possible on real data because true matches are unknown); and (c) the generated data, as well as the generator program itself, can be published.

This manual describes a flexible data generator developed in 2012 by the Australian National University (ANU) in collaboration with Fujitsu Laboratories. The aim of this generator is to improve upon earlier data generator systems that were developed at the ANU as part of the *Febrl* (Freely Extensible Biomedical Record Linkage) [4, 3] project. The specific objectives of this improved data generator are:

- The generator allows the flexible specification of different attribute types:
  - Attributes that generate values based on frequency look-up files that are provided by the user.
  - Attributes that generate values based on a function provided by the user.
  - Compound attributes, where the value generated for one attribute depends upon the value generated in the other attributes(s). The values in these attributes will be based either on look-up tables or functions, as appropriate. Specifically, the following four types of compound attributes can be generated:
    1. An attribute with continuous values that are dependent on the values in one categorical attribute (for example, blood pressure depending upon gender).
    2. An attribute with continuous values that are dependent on the values in another continuous attribute (for example, blood pressure depending upon weight).
    3. An attribute with categorical values that are dependent on the values in another categorical attribute (for example, city of residence depending upon gender).
    4. An attribute with continuous values that are dependent on the values in two other categorical attributes (for example, blood pressure depending upon gender and city of residence).
- The generator allows the specification of more than one attribute of each of the above described attribute and compound attribute types.
- The names, order and number of specified attributes determine the structure of the data set that is generated.

- The number of ‘original’ and ‘duplicate’ (modifications of the original records) can be specified by the user.
- The generator allows different types of modifications to be applied to the generated records. Specifically, it will allow
  1. random character edits of values,
  2. random exchange of a value from a frequency look-up file with another value from the same look-up file,
  3. random application of optical character recognition modifications, and
  4. random application of phonetic modifications.
- The generator allows data encoded in different Unicode character sets to be generated.

The generated data sets are written into text files in the comma separated values (CSV) format, which allows easy import into spreadsheet programs and database systems. Each generated record is given a unique identifier value in the first attribute (column) of a generated data set. The following shows a small example of generated data.

```
rec-number,gender,given-name,surname,postcode,city,telephone-number,credit-card-number
rec-00-org,female,katelyn,krzyszton,2604,sydney,07 4614 9969,6209 8159 5895 6125
rec-00-dup-0,female,katelyn,krzyszdon,missing,sydney,,6209 859 5895 6125
rec-00-dup-1,female,kaelyn,krzywzton,26p4,sydney,,6209 8159 5895 6125
rec-01-org,male,ruby,alderson,2914,sydney,02 5091 1848,7334 2846 5794 9515
rec-01-dup-0,,rupy,alderson,291e,sysney,,7334 2846 5794 9515
rec-01-dup-1,,rupy,alderson,2914,sydneyg,,7334 28464 5794 9515
rec-02-org,female,katherine,sukwatthananan,2906,perth,08 7387 6792,6629 1891 7867 8126
rec-02-dup-0,,kadherine,jukwatthananan,29o6,,08 7387 6792,6629 1891 7867 8126
rec-03-org,female,alexandra,white,2602,canberra,02 3729 9751,4657 5912 6390 0301
rec-03-dup-0,female,alexantra,whide,2602,kanberra,,4657 5912 6390 7301
rec-03-dup-1,female,alezxandra,hite,2602,,02 3729 9751,4657 59123 6390 0301
rec-03-dup-2,,alexantra,whit,missing,canberra,02 3729 9751,4657 5912 63900301
```

The remainder of this manual provides details about the data generator programs and how they can be configured, the structure of look-up tables that are used to generate attribute values, how the generator programs can be installed, and details about the testing procedures applied.

Specifically, the following chapter provides an overview of the structure of the modules that constitute the data generator. Chapter 3 then details each of the classes provided by the generator that allow data of different types and with different characteristics to be generated and modified. The formats of the different types of look-up files used are presented in Chapter 4, and in Chapter 5 an example generator program is explained in detail to allow the user to understand the different components that are required to generate data. Installation procedures are then covered in Chapter 6, and testing procedures in Chapter 7. Finally, Chapter 8 contains a discussion of how the generator modules and look-up files will need to be modified to allow the generation of data sets with different Unicode character sets.

The appendix at the end of this document contains a copy of the **Mozilla Public License Version 2** under which this software is licensed.

Note that a separate document, **api.pdf** is provided with this software which includes the details of the Application Programming Interface (API) of all modules of the data generator, including details of all classes, methods and their input arguments.

Additional modules of the this data generator, that will allow the generation of longitudinal (temporal) data sets, as well as data sets that contain not just records about individuals but also families and households, will be added in the near future. Work on some of these additional modules is currently (March 2012) being carried out as part of computer science student projects at the Australian National University.

CHAPTER

TWO

MODULE OVERVIEW

The data generator system consists of several modules, each containing various classes and methods, or individual functions that are not part of a class. Figure 2.1 shows an overview of the module structure of the data generator, and which modules need to be modified or extended by the user in order to generate data with different characteristics.

In this chapter, an overview of the functionality of each module is provided. The following chapter then describes the classes, methods and functions of each module in more detail. The input arguments and descriptions of the expected values of all methods and functions, known as the Application Programming Interface (API), are given in a separate document, named **api.pdf**, that is included with this software.

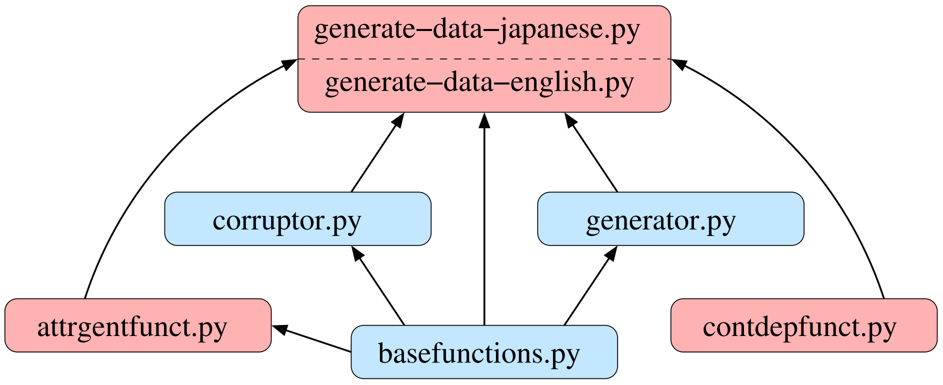

Figure 2.1: Overview of the data generator module structure. Modules shown in red (dark grey) contain functionalities or parameters that need to be modified or extended by the user, while modules shown in blue (light grey) do not need to be modified. Arrows indicate which module (at a lower level) is imported by which other module (at a higher level).

Starting at the lowest level, the `basefunctions.py` module contains functions that allow checking of various values and types of variables. These functions are used by other modules to validate the input arguments given to their functions and methods. The `basefunctions.py` module also contains functions related to input and output, such as for reading and writing text files in the comma separated values (CSV) format, and for formatting numbers into strings. While there is generally no need for a user to modify this module, one function within this module (`char_-set_ascii`) will need to be considered when data in different Unicode encodings are to be generated. This will be further discussed in Chapter 8.

The module `attrgentfunct.py` includes functions that generate values of attributes based on some random processes. Specifically, functions given in this module are used by the `GenerateFuncAttribute` class in the `generator.py` module. The requirements of a function that is used to generate attribute values is that it must return string values and that between zero and five input arguments can be handled. The present version of the `attrgentfunct.py` module contains several example functions that can generate Australian telephone numbers, credit card numbers, and numerical values and age values that follow a normal or uniform distribution according to the argument values set by the user. Users can add as many functions as they require into this module and thereby build their own library of attribute generation functions according to their needs.

The `contdepfunct.py` module also contains functions that are used in the data generation process. These functions are specifically used in the class `GenerateContContCompoundAttribute` available in the `generator.py` module, which can generate a pair of continuous attributes where the second attribute depends upon the value of the first attribute. A function given in the `contdepfunct.py` module is assumed to be used to generate numerical (floating-point) values for the second of these two continuous attributes. The requirement of each such function is to have one input argument which is assumed to be the numerical value of the first continuous attribute, and they must return a numerical value which is assumed to be the value of the second continuous attribute. Users can again add as many functions as they require into this module and thereby build their own library of continuous attribute generation functions according to their needs.

The main `generator.py` module contains several classes that allow the generation of records according to parameter settings provided by a user. The details of each class in this module will be described in the following chapter. Two types of classes are given. The first allows a user to specify how the values of a single attribute or a compound attribute (two or three attributes that depend upon each other) will be generated. The second type of class is the main class that generates the 'original' records according to the settings of the different attribute generation methods defined. In Chapters 3 and 5 examples of these methods will be given.

The second main module, `corruptor.py`, contains the classes that allow the modification (corruption) of attribute values taken from the 'original' records to generate 'duplicate' records. Again two types of classes are included in this module. The first type of class allows the specification of different types of modifications, while the second type of class allows the specification of how modifications are to be introduced into records, and how many such 'duplicate' records are to be generated. Examples of how the corruption methods are to be employed are given in Chapters 3 and 5.

The top level modules of the data generator system are the modules where a user specifies how a certain synthetic data set is to be generated. Two example modules are provided with this software, one module (named `generate-data-english.py`) to generate English data sets, while the second (named `generate-data-japanese.py`) can generate example data sets in a Japanese Unicode encoding. Various parameters are to be set by the user, as will be described further in Chapter 5.

Cautionary notes:

Throughout this manual, various classes and look-up file formats will be described that either require probability values or frequency counts of categorical values to be specified. The following general assumptions hold:

- Probability values must be given as numerical values between 0.0 and 1.0. For certain methods and parameter definitions several such probabilities are required, and in some of these definitions the sum of the given probabilities must be 1.0. If such a requirement is needed it will be described appropriately.
- The counts given in look-up tables to set the frequency of occurrences of categorical values must be positive integer values. In the examples shown throughout this manual, the sum of these counts is 100 in order to facilitate easier understanding of the relationship between frequency counts and the likelihood that a categorical value will be selected as an attribute value during the data generation process. This summing to 100 is only for illustrative purposes and not necessary when frequency counts of real data are being used.

For all modules of the data generator software, all input attributes of all functions and methods, as well as the elements in look-up files, are checked for both their correct type (string, numerical, etc.) and if their value is in the correct range. If any wrong or out-of-range value is encountered then the program will stop with an exception message that will provide details about why an exception occurred. This will help the user to fix the wrong input parameter value or wrong data element in a look-up file.

CHAPTER

THREE

DETAILS OF CLASSES AND METHODS

This chapter provides details of the functions, classes and their methods in the modules `basefunctions.py`, `attrgentfunct.py`, `contdepfunct.py`, `generator.py`, and `corruptor.py`. The details of how to use the functionality of these modules in a top-level module such as `generate-data-english.py` or `generate-data-japanese.py` when generating a data set is then given in Chapter 5.

Several standard Python modules are required by some of the data generator modules. These standard modules are included in any Python distribution. The required modules are: `codecs`, `math`, `os`, `random`, and `type`. Detailed information about these modules is available from the Python Web site at http://www.python.org/modules/index/ .

All modules of the data generator contain documentation embedded into the code, and the Python `help()` function can be used to view these documentation texts. Assuming a Python interpreter has been started in the main data generator folder/directory, documentation about the `generator.py` module and its classes can for example be viewed using the following instructions:

```
>>> import generator
>>> help(generator)
>>> help(generator.GenerateFuncAttribute)
>>> help(generator.GenerateDataSet)
```

3.1 Module `basefunctions.py`

This module contains the following functions which are required by the other modules of the data generator:

- `check_is_`*X*`(variable, value)`

  These functions (for different tests *X*, such as *string*, *number*, *positive* etc.) check if the given input `value` is of the correct type and/or in the expected range. If this is not the case an exception will be triggered and the program aborted. The `variable` needs to be a string which gives the name of the variable for which the value is checked.

- `check_unicode_encoding_exists(unicode_encoding_str)`

  This function checks if the provided Unicode encoding (given as a string) is a known encoding, which means it is listed in the standard encodings table available on the Python Web site at http://docs.python.org/library/codecs.html#standard-encodings .

- `char_set_ascii(s)`

  This function determines if the given input string `s` contains (a) letters only, (b) digits only, (c) letters and whitespaces, (d) digits and whitespaces, (e) letters and digits, or (f) letters, digits, and whitespaces. Depending upon the content of the input string, the function returns a string that contains the corresponding range of characters. For example, in case (b) it returns the string ‘`0123456789`’, while in case (c) it returns ‘ `abcdefghijklmnopqrstuvwxyz`’.

This function is used by the `CorruptValueEdit` class in the `corruptor.py` module to determine what characters can be used for the insert or substitution edit operations.

If character sets other than ASCII are used to generate and corrupt a data set then a function similar to this function needs to be provided by the user. Such a function simply has to return a string that contains all possible character values that can be used for insert and substitution edit operations.

- `float_to_str(f, format_str)`

  This function converts a given numerical value `f` into a string according to the format string given. Possible format string values are:

  - `int` Return the numerical value `f` converted into a string as integer value.
  - `float1` Returns the numerical value `f` converted into a string as floating-point value with one digit.
  - `float2` Returns the numerical value `f` converted into a string as floating-point value with two digits.
  - `...`
  - `float9` Returns the numerical value `f` converted into a string as floating-point value with nine digits.

- `str2comma_separated_list(s)`

  A function which splits the string `s` at each comma character found in `s` into a list of sub-strings. For each sub-string, if it starts and ends with the same quote character, either single or double quotes, then these quote characters will be removed.

- `read_csv_file(file_name, encoding, header_line)`

  With this function, text files in the Comma Separated Values (CSV) format can be loaded. The `file_name` has to point to an existing CSV file, and the `encoding` must be a valid Unicode encoding string (see also the `check_unicode_encoding_exists` function above). The argument `header_line` is a flag that has to be set to `True` or `False`. If set to `True`, it is assumed that the first line in the file corresponds to a header line which is loaded specifically.

  The CSV file can contain comment lines, which are lines that start with a '`#`' character, or empty lines. Both these types of lines will be skipped when the file is read.

  This function returns two values, the first being the header line (or `None` in case no header line was specified), while the second value is a list that contains the actual rows (or lines) of the file, one list per row.

- `write_csv_file(file_name, encoding, header_line, file_data)`

  This last function in the `basefunctions.py` module is used to write a generated data set into a CSV file with the given `file_name`. If this file already exists it will be overwritten.

  The header line can be a list containing the names of the attributes of the data set that is being written, in which case these names will become the first line in the file. If no header line is to be written then the argument `header_line` can be set to `None`. The Unicode encoding string again must correspond to a valid Unicode encoding. The argument `file_data` contains the data to be written into the file. This must be a list containing lists, each of which will become one row (line) in the output CSV file. The number of elements in each of these lists must be the same as the number of elements in the header line (if a header line is provided).

3.2 Module `attrgenfunct.py`

This module contains the example functions that generate values of attributes based on some random process. These functions are used by the `GenerateFuncAttribute` class in the `generator.py` module to generate attribute values.

The user can add new functions to this module to extend the functionality of the data generator. The two requirements of any function that is to be used by the `GenerateFuncAttribute` class are (a) the function can have between zero and five input arguments, and (b) the function must return a string.

The `attrgenfunct.py` module contains the following functions:

- `generate_phone_number_australia()`

  This function randomly generates strings in the format of Australian telephone numbers made of a two-digit area code and two blocks of four-digit numbers, separated by whitespaces. Example values are: '`03 7920 9594`', '`02 3729 9751`', and '`04 8694 4929`'.

- `generate_credit_card_number()`

  This function randomly generates strings in the format of credit card numbers made of four blocks of four-digit numbers separated by whitespaces. Example values are: '`4303 1239 7920 9594`', '`3202 0758 3729 9751`', and '`9804 4438 8694 4929`'

- `generate_uniform_value(min_val, max_val, val_type)`

  This function randomly generates numerical values that are uniformly distributed between a lower boundary `min_val` and an upper boundary `max_val`. The argument `val_type` determines the format of the returned string, it can be set to one of the format string values described with the `float_to_str` function from the `basefunctions.py` module.

- `generate_uniform_age(min_val, max_val)`

  This function returns randomly generated (with uniform distribution) age values in the range between `min_val` (which as to be 0 or larger) and `max_val` (which as to be 130 or smaller), and returns them as integer string values.

- `generate_normal_value(mu, sigma, min_val, max_val, val_type)`

  This function randomly generates numerical values that are normally distributed with an average value of `mu` and a standard deviation of `sigma`. Lower and upper boundaries can be given optionally by setting the `min_val` and/or `max_val` arguments to a numerical value. If either of these two arguments is set to `None`, then no lower and/or upper boundary will be enforced.

  It is possible to set one of these two arguments to `None` and the other to a numerical value depending upon the type and distribution of values one is interested to be generated. For example, when normally distributed blood pressure values are to be created then a lower boundary of zero (`min_val = 0)` would be appropriate.

  The argument `val_type` again determines the format of the returned string, it can be set to one of the format string values described with the `float_to_str` function from the `basefunctions.py` module.

- `generate_normal_age(mu, sigma, min_val, max_val)`

  This function returns randomly generated (with normal distribution) age values with a mean of `mu` and a standard deviation of `sigma` in the range between `min_val` (which as to be 0 or larger) and `max_val` (which as to be 130 or smaller), and returns them as integer string values.

3.3 Module `contdepfunct.py`

This module contains two example functions that generate numerical values for continuous attributes that are dependent upon an input value (also numerical). These functions can be used by the class `GenerateContContCompoundAttribute` in the `generator.py` module.

The requirement of such functions is that they have as input an argument that is assumed to be numerical values, and that they also return a numerical value.

The following two example functions are included in the `contdepfunct.py` module:

- `blood_pressure_depending_on_age(age)`

This function randomly generates a numerical blood pressure value that depends upon the given `age` value. It is assumed that for a given age value the blood pressure is normally distributed with an average blood pressure of 75 at birth (age 0) and of 90 at age 100. The standard deviation used for this normal distribution is 4.

- `salary_depending_on_age(age)`

  This function randomly generates a numerical salary value that depends upon the given `age` value. It is assumed that for a given age value the salary is uniformly distributed with a minimum salary starting at $10,000$ independent of the age value, and a maximum salary value that is set as $20,000$ at age 18 and that increases to $150,000$ at age 60. For `age` values below 18 a salary value of 0 will be returned.

3.4 Module `generator.py`

This module contains several classes that allow the generation of different types of attribute values, as well as the generation of data sets consisting of 'original' records. A user can customise the data generation process by using different such data generation objects according to the type of data they aim to generate.

The first two classes are to be used for individual attributes where the values are generated independently from all other attributes, while the remaining four classes generate two or even three attributes with some form of dependency in the way attribute values are generated.

- `GenerateFreqAttribute`

  This class specifies the generation of an attribute where values (assumed to be strings) are loaded from a look-up file such that the likelihood of a certain value being generated depends upon the frequency (or count) of the value in the look-up file. The higher the frequency or count for a value is the more likely it will be generated. The format of such frequency look-up files is presented in detail in the following chapter.

  The following input arguments need to be given when an attribute of this class is to be generated:

  - `attribute_name`
    The name of this attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. Each attribute has to be given a unique name.
  - `freq_file_name`
    The name of the file which contains the attribute values and their frequencies. The format of such files is presented in detail in the following chapter.
  - `has_header_line`
    A flag which needs to be set to `True` if the first line in the frequency look-up file corresponds to a header line (i.e. does not contain an attribute value), or to `False` if there is no header line in the frequency look-up file.
  - `unicode_encoding`
    The Unicode encoding (a string name) used for encoding the values in the frequency look-up file.

  An example (taken from the `generate-data-english.py` module) of how to generate a frequency attribute follows.

```
surname_name_attr = \
    generator.GenerateFreqAttribute(attribute_name = 'surname',
                freq_file_name = 'lookup-files/surname-freq.csv',
                has_header_line = False,
                unicode_encoding = 'ascii')
```

- `GenerateFuncAttribute`

  This class specifies the generation of an attribute where values (assumed to be strings) are generated based on a function provided by the user. Such a function has to generate string values according to some process defined by the user. Examples of such functions are available in the module `attrgenfunct.py` which has been described previously.

  The following input arguments need to be given when an attribute of this class is to be generated:

  – `attribute_name`
    The name of this attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. Each attribute has to be given a unique name.
  – `function`
    A reference to a Python function which will be used to generate attribute values. The requirements of this function are that it has to return string values, and that it can have between zero and five input arguments (which have to be specified in the following `parameters` argument).
  – `parameters`
    A list of one or more parameters (maximum 5) that will be passed to the `function` when it is called during the value generation process. The default value for this argument is `None`, which means that the `function` does not require any input arguments.

  Two examples (taken from the `generate-data-english.py` module) of how to generate a function based attribute follow.

```
credit_card_attr =  \
    generator.GenerateFuncAttribute(attribute_name = 'credit-card-number',
                function = attrgenfunct.generate_credit_card_number)

income_normal_attr = \
    generator.GenerateFuncAttribute(attribute_name = 'income-normal',
                function = attrgenfunct.generate_normal_value,
                parameters = [50000, 20000, 0, 1000000, 'float2'])
```

- `GenerateCateCateCompoundAttribute`

  This class specifies the generation of two attributes that contain categorical values, where a value generated for the second categorical attribute depends upon the value generated for the first categorical attribute. This class allows for example the generation of an attribute city of residence that depends upon the value of a gender attribute (resulting in different distributions of males and females for different cities), or of medication names that depend upon the gender value.

  The details of categories and their distributions need to be provided in a look-up file that has a specific format, as will be described in the following chapter.

  The following input arguments need to be given when a compound set of two attributes of this class is to be generated:

  – `categorical1_attribute_name`
    The name of the first categorical attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the second categorical attribute, and also different from the names of all other attributes that are to be generated.
  – `categorical2_attribute_name`
    The name of the second (the dependent) categorical attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator

program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the first categorical attribute, and also different from the names of all other attributes that are to be generated.

- `lookup_file_name`
  The name of the file which contains the attribute values and their counts (frequencies) for both categorical attributes. The format of such files is presented in detail in the following chapter.
- `has_header_line`
  A flag which needs to be set to `True` if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to `False` if there is no header line in the look-up file.
- `unicode_encoding`
  The Unicode encoding (a string name) used for encoding the values in the look-up file.

An example (taken from the `generate-data-english.py` module) of how to generate a compound set of two categorical attributes follows.

```
gender_city_compound_attr = \
    generator.GenerateCateCateCompoundAttribute(\
                categorical1_attribute_name = 'gender',
                categorical2_attribute_name = 'city',
                lookup_file_name = 'lookup-files/gender-city.csv',
                has_header_line = True,
                unicode_encoding = 'ascii')
```

- `GenerateCateContCompoundAttribute`

  This class specifies the generation of two attributes, the first containing categorical values and the second containing numerical values that depend upon the categorical values of the first attribute. This class allows for example the generation of an attribute that contains salary values that depend upon the value of a gender attribute, or of blood pressure values that depend upon gender values.

  The details of categories and the functions and their parameters used for the generation of numerical values need to be provided in a look-up file that has a specific format, as will be described in the following chapter.

  The following input arguments need to be given when a compound set of two attributes of this class is to be generated:

  - `categorical_attribute_name`
    The name of the categorical attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the continuous attribute, and also different from the names of all other attributes that are to be generated.
  - `continuous_attribute_name`
    The name (as a string) of the continuous attribute that is dependent on the categorical attribute. This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the categorical attribute, and also different from the names of all other attributes that are to be generated.
  - `lookup_file_name`
    The name of the file which contains the attribute values and their counts (frequencies) for the categorical attribute, and the functions and their parameters used for the generation of values in the continuous attribute. The format of such files is presented in detail in the following chapter.
  - `has_header_line`
    A flag which needs to be set to `True` if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to `False` if there is no header line in the look-up file.

- `unicode_encoding`
  The Unicode encoding (a string name) used for encoding the values in the look-up file.
- `continuous_value_type`
  This argument determines the format of the values generated in the continuous attribute, as they are converted from numbers into strings. This argument can be set to one of the format string values described with the `float_to_str` function from the `basefunctions.py` module, which are: '`int`', '`float1`','`float2`', `...`, '`float9`'.

An example (taken from the `generate-data-english.py` module) of how to generate a compound set made of a categorical and a continuous attribute follows.

```
state_income_compound_attr = \
    generator.GenerateCateContCompoundAttribute(\
                categorical_attribute_name = 'state',
                continuous_attribute_name = 'income',
                continuous_value_type = 'float1',
                lookup_file_name = 'lookup-files/state-income.csv',
                has_header_line = False,
                unicode_encoding = 'ascii')
```

- `GenerateContContCompoundAttribute`

  This class specifies the generation of two continuous attributes, where values in the second attribute depend upon the values in the first attribute. This class allows for example the generation of a salary attribute where the values depend upon the values in an age attribute, or of an attribute containing blood pressure values that depend upon age values.

  While any function that generates numerical values can be used for the second (dependent) attribute, the current version of the data generator can only generate values for the first continuous attribute that are either uniformly or normally distributed according to parameters provided by the user.

  The following input arguments need to be given when a compound set of two attributes of this class is to be generated:

  - `continuous1_attribute_name`
    The name of the first continuous attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the second continuous attribute, and also different from the names of all other attributes that are to be generated.
  - `continuous2_attribute_name`
    The name of the second (the dependent) continuous attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the first continuous attribute, and also different from the names of all other attributes that are to be generated.
  - `continuous1_funct_name`
    The name of the function that is to be used when randomly generating continuous attribute values in the first attribute. Currently two possible function types are implemented: '`uniform`' and '`normal`'.
  - `continuous1_funct_param`
    A list containing the parameters that are required for the function that generates the continuous values in the first attribute. These parameters are similar to the parameters that are required for the functions `generate_uniform_value` and `generate_normal_value` that were described earlier in the section covering the `attrgenfunct.py` module.

For a 'uniform' function, two parameters are required: [min_val, max_val]. They correspond to the minimum and maximum values that can be generated.

For a 'normal' function, four parameters are required: [mu, sigma, min_val, max_val]. The first corresponds to the mean of the normal distribution that is to be generated, while the second parameter will be its standard deviation. The third and fourth parameters are optional lower and upper boundaries that can be set to a numerical value, or be set to None in which case no lower and/or upper boundary of values will be enforced.

- continuous2_function
  Similar to the GenerateFuncAttribute class described earlier, this argument must be a reference to a Python function which will be used to generate the values in the second attribute. The requirements of this function are that it has to return a numerical (floating-point) value, and that it needs to have one input argument which is assumed to be the numerical values generated for the first continuous argument. Examples of such functions are provided in the contdepfunct.py which was described above.
- continuous1_value_type
  This argument determines the format of the values generated in the first continuous attribute, as they are converted from numbers into strings. This argument can be set to one of the format string values described with the float_to_str function from the basefunctions.py module, which are: 'int', 'float1','float2', ..., 'float9'.
- continuous2_value_type
  This argument determines the type of values returned for the second attribute, similar to the previous argument.

An example (taken from the generate-data-english.py module) of how to generate a compound set of two continuous attributes follows.

```
age_blood_pressure_compound_attr = \
    generator.GenerateContContCompoundAttribute(\
                continuous1_attribute_name = 'age',
                continuous2_attribute_name = 'blood-pressure',
                continuous1_funct_name =     'uniform',
                continuous1_funct_param =    [10,110],
                continuous2_function = \
                    contdepfunct.blood_pressure_depending_on_age,
                continuous1_value_type = 'int',
                continuous2_value_type = 'float3')
```

- GenerateCateCateContCompoundAttribute

  This class specifies the generation of three attributes, where the first two attributes will contain categorical values and the third attribute will contain continuous values. The values in the second categorical attribute will depend upon the values in the first categorical attribute, while the values in the continuous attribute then depend upon the values in the second (and therefore also the first) categorical attribute. This class allows for example the generation of an attribute that contains numerical blood pressure values that depend upon gender and city of residence values, or of salary values that depend upon gender and profession values.

  The details of categories and the functions and their parameters used for the generation of numerical values need to be provided in a look-up file that has a specific format, as will be described in the following chapter.

  The following input arguments need to be given when a compound set of three attributes of this class is to be generated:

  - categorical1_attribute_name
    The name of the first categorical attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify

the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the second categorical attribute and from the continuous attribute name, and also different from the names of all other attributes that are to be generated.

- `categorical2_attribute_name`
  The name of the second (the dependent) categorical attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the name of the first categorical attribute and from the continuous attribute name, and also different from the names of all other attributes that are to be generated.
- `continuous_attribute_name`
  The name of the continuous attribute (as a string). This value will be written into the header line of the generated output file (if so specified), and this name is used in a data generator program to identify the attribute, as will be shown in Chapter 5. This name needs to be different from the names of both categorical attributes, and also different from the names all other attributes that are to be generated.
- `lookup_file_name`
  The name of the file which contains the attribute values and their frequencies for both categorical attributes, as well as the function names and parameters to be used for the continuous attribute. The format of such files is presented in detail in the following chapter.
- `has_header_line`
  A flag which needs to be set to `True` if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to `False` if there is no header line in the look-up file.
- `unicode_encoding`
  The Unicode encoding (a string name) used for encoding the values in the look-up file.
- `continuous_value_type`
  This argument determines the format of the values generated in the continuous attribute, as they are converted from numbers into strings. This argument can be set to one of the format string values described with the `float_to_str` function from the `basefunctions.py` module, which are: '`int`', '`float1`','`float2`', `...`, '`float9`'.

An example (taken from the `generate-data-english.py` module) of how to generate a compound set of two categorical and one continuous attribute follows.

```
sex_town_salary_compound_attr = \
    generator.GenerateCateCateContCompoundAttribute(\
                categorical1_attribute_name = 'sex',
                categorical2_attribute_name = 'town-name',
                continuous_attribute_name = 'salary',
                continuous_value_type = 'float4',
                lookup_file_name = 'lookup-files/gender-town-salary.csv',
                has_header_line = False,
                unicode_encoding = 'ascii')
```

- `GenerateDataSet`

  This class provides a mechanism to specify how the different attribute generation objects are to be combined and how the 'original' records of a data set are to be generated.

  The following input arguments need to be given when a data set generation object is initialised (constructed):

  - `output_file_name`
    The name of the file that will be generated. This will be a comma separated values (CSV) file. If the given file name does not end with the extension '`.csv`' then this extension will be added.

  - `write_header_line`
    A flag that can be set to `True` or `False` which determines if a header line with the attribute names is to be written at the beginning (first row) of the output file or not. The default for this argument is `True`.
  - `rec_id_attr_name`
    The name of the record identifier attribute. Values in this attribute are generated automatically during the generation process, such that every generated record will be given a unique identifier. This name determines how this attribute will be named in the header line. This name must be different from the names of all other attributes that are being generated.
  - `number_of_records`
    The number of 'original' records that are to be generated. This corresponds to the number of records that are generated during the `generate` process.

    Note that this number is not the total number of records that will be generated. The total number of records corresponds to the sum of the number of 'original' and the number of 'duplicate' records that are generated (the latter will be discussed further in the section below detailing the methods of the `corruptor.py` module).
  - `attribute_name_list`
    This argument is the list of attributes that are to be generated for each record, and the sequence of how they are to be written into the output file. The argument is a list containing the names of attributes, for which an attribute generation method (as presented above) has been initialised. The attribute names given in this list will become the header line of the output file (if a header line is to be written).
  - `attribute_data_list`
    This argument is a list which contains the actual attribute generation methods (objects) that have been initialised. Note that the order of elements in this list does not determine the order of attributes (columns) in the output files. This list can also include attribute generation methods that have been defined but that are not written into the output file.
  - `unicode_encoding`
    The Unicode encoding (a string name) of how the output file will be encoded.

An example of how to initialise a `GenerateDataSet` object is provided in Chapter 5.

The `GenerateDataSet` class has two main methods which will be used to generate the 'original' records (called `generate`), and to write the generated records into the specified output file (named `write`). An example of how these two methods are used is also given in Chapter 5.

3.5 Module `corruptor.py`

This module contains several classes that allow the modification of attribute values that were generated by the classes provided in the `generator.py` module. Another class facilitates the overall modification of 'original' records into 'duplicate' records. A user can customise the data modification process by using different corruption objects according to the type of modifications they aim to generate.

- `position_mod_uniform(in_str)`

  This function randomly selects with a uniform distribution a position between 0 and the length (in characters) of the given input string `in_str`. It can be used to select the position of where a modification in a string is to be applied.

  If the given input string `in_str` is empty then a position value of 0 will be returned.

- `position_mod_normal(in_str)`

  This function randomly selects with a normal distribution a position between 0 and the length (in characters) of the given input string `in_str`. It can be used to select the position of where a modification in a string is to be applied.

The mean value of the used normal distribution is set to a position one character behind half the length of the input string, leading to higher probabilities that a position in the middle or end of the string will be returned. This is based on studies on the distribution of errors in real name strings which have shown that errors such as typographical mistakes are more likely to appear towards the middle and end of a string but not at the beginning [1, 5].

If the given input string `in_str` is empty then a position value of 0 will be returned.

- `CorruptMissingValue`

  This class provides a mechanism on how a given attribute value is to be replaced with a specific string that indicates the value is missing.

  The following input argument needs to be given when such a corruptor object is initialised (constructed):

  - `missing_val`
    The string which designates a missing value. The default value is the empty string ‘’.

  An example (taken from the `generate-data-english.py` module) of how to initialise corruptor objects for two different missing value strings is given below.

```
missing_val_corruptor = corruptor.CorruptMissingValue()

postcode_missing_val_corruptor = corruptor.CorruptMissingValue(\
             missing_val='missing')
```

- `CorruptValueEdit`

  This class provides a mechanism to modify attribute values through four types of single character edits: insert a character, delete a character, substitute a character with another character from a specified range, and transposing two adjacent characters.

  The following input arguments need to be given when such a corruptor object is initialised (constructed):

  - `position_function`
    A reference to a Python function which determines the location within a string value of where a modification (corruption) is to be applied. This can for example be one of the two position functions described above, or it can be a function provided by the user.

    The requirements of such a function are that its input is a string and that it returns a positive integer value that is in the range of the length of the given input string.
  - `char_set_funct`
    A reference to a Python function which determines the set of characters that can be inserted or used of substitution. This can for example be the function `char_set_ascii` which was described in the section on the `basefunctions.py` above, or a function provided by the user.

    The requirements of such a function are that its input is a string and that it returns a string that contains all possible characters that can be used when substituting or inserting a character if the corresponding edit operation is carried out.
  - `insert_prob`
    The probability (between 0.0 and 1.0) that an insert operation (of a single character) is applied to an attribute value.
  - `delete_prob`
    The probability (between 0.0 and 1.0) that a delete operation (of a single character) is applied to an attribute value.
  - `substitute_prob`
    The probability (between 0.0 and 1.0) that a substitution operation (of a character by another character) is applied to an attribute value.

- `transpose_prob`
  The probability (between 0.0 and 1.0) that a transposition operation (of two adjacent characters) is applied to an attribute value.

Important: The sum of the four probabilities `insert_prob`, `delete_prob`, `substitute_prob`, and `transpose_prob` has to be 1.0.

An example (taken from the `generate-data-english.py` module) of how to initialise a corruptor with uniformly distributed edit probabilities is given below.

```
uniform_edit_corruptor = corruptor.CorruptValueEdit(\
            position_function = corruptor.position_mod_normal,
            char_set_funct = basefunctions.char_set_ascii,
            insert_prob = 0.25,
            delete_prob = 0.25,
            substitute_prob = 0.25,
            transpose_prob = 0.25)
```

- `CorruptValueKeyboard`

  This class provides a mechanism to modify attribute values through errors that mimic typing mistakes on a keyboard. Specifically, mistakes of neighbouring keys (in the same row or the same column) are modelled.

  The current version of this class has the keyboard layout (assumed to be a QWERTY keyboard) hard-coded in the implementation as two data structures of neighbouring row and column keys. However, a user can easily modify these hard-coded data structures to adjust them to other keyboard layouts.

  When an attribute value is modified, a character in the value will be randomly selected using the `position_-function`, and then according to the row and column probabilities given the character will be replaced by a neighbouring keyboard character in the same row or the same column.

  The following input arguments need to be given when such a corruptor object is initialised (constructed):

  - `position_function`
    A reference to a Python function which determines the location within a string value of where a modification (corruption) is to be applied. This can for example be one of the two position functions described above, or it can be a function provided by the user.

    The requirements of such a function are that its input is a string and that it returns a positive integer value that is in the range of the length of the given input string.
  - `row_prob`
    The probability (between 0.0 and 1.0) that a modification occurs in the same row of the keyboard, i.e. the probability that a character is being replaced with the character of a neighbouring key in the same row (left or right).
  - `col_prob`
    The probability (between 0.0 and 1.0) that a modification occurs in the same column of the keyboard, i.e. the probability that a character is being replaced with the character of a neighbouring key in the same column (above or below).

  Important: The sum of the two probabilities `row_prob` and `col_prob` has to be 1.0.

  An example (taken from the `generate-data-english.py` module) of how to initialise a keyboard corruptor where row typing mistakes occur double as likely as column typing mistakes is given below.

```
keyboard_corruptor = corruptor.CorruptValueKeyboard(\
            position_function = corruptor.position_mod_uniform,
            row_prob = 0.667,
            col_prob = 0.333)
```

- `CorruptValueOCR`

  This class provides a mechanism to modify attribute values through errors that mimic Optical Character Recognition (OCR) variations. Such variations are loaded from a look-up file that contains character sequences (made of between one and up-to three characters) and their corresponding OCR variations. The format of this look-up file will be described in detail in the following chapter. Example modifications include '`s`' and '`5`', '`g`' and '`q`', or '`m`' and '`rn`'. Note that the entries in a look-up file are handled symmetric, for example both modifications from '`m`' to '`rn`' and the other way round from '`rn`' to '`m`' are considered.

  The following input arguments need to be given when such a corruptor object is initialised (constructed):

  – `position_function`
    A reference to a Python function which determines the location within a string value of where a modification (corruption) is to be applied. This can for example be one of the two position functions described above, or it can be a function provided by the user.

    The requirements of such a function are that its input is a string and that it returns a positive integer value that is in the range of the length of the given input string.

  – `lookup_file_name`
    The name of the file which contains the OCR variations as pairs of original character sequence and modified character sequence. The format of such files is presented in detail in the following chapter.

  – `has_header_line`
    A flag which needs to be set to `True` if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to `False` if there is no header line in the look-up file.

  – `unicode_encoding`
    The Unicode encoding (a string name) used for encoding the values in the look-up file.

  An example (taken from the `generate-data-english.py` module) of how to initialise an OCR corruptor is given below.

```
ocr_corruptor = corruptor.CorruptValueOCR(\
            position_function = corruptor.position_mod_normal,
            lookup_file_name = 'lookup-files/ocr-variations.csv',
            has_header_line = False,
            unicode_encoding = 'ascii')
```

- `CorruptValuePhonetic`

  This class provides a mechanism to modify attribute values through errors that mimic phonetic mistakes. Such modifications are loaded from a look-up file that contains character sequences and their corresponding phonetic variations. The format of this look-up file will be described in detail in the following chapter. A detailed description of this approach to modifying attribute values is provided in [5].

  The following input arguments need to be given when such a corruptor object is initialised (constructed):

  – `lookup_file_name`
    The name of the file which contains the phonetic modification patterns as tuples of original character sequence and modified character sequence and conditions that govern when a modification can be applied. The format of such files is presented in detail in the following chapter.

  - has_header_line
    A flag which needs to be set to True if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to False if there is no header line in the look-up file.
  - unicode_encoding
    The Unicode encoding (a string name) used for encoding the values in the look-up file.

  Note that the position_function is not required by this corruptor method because many of the phonetic modification patterns can only be applied in certain positions of a value.

  An example (taken from the generate-data-english.py module) of how to initialise a phonetic corruptor is given below.

```
phonetic_corruptor = corruptor.CorruptValuePhonetic(\
            lookup_file_name = 'lookup-files/phonetic-variations.csv',
            has_header_line = False,
            unicode_encoding = 'ascii')
```

- CorruptCategoricalValue

  This class provides a mechanism to modify categorical attribute values by using known variations such as misspellings of known values, for example misspellings or nicknames for given names or surnames. Such variations are loaded from a look-up file that contains attribute values and their misspellings. The format of this look-up file will be described in detail in the following chapter.

  It is possible for a given known categorical attribute value to have more than one variation, in which case one of them will be randomly selected during the corruption process.

  The following input arguments need to be given when such a corruptor object is initialised (constructed):

  - lookup_file_name
    The name of the file which contains the categorical values and their variations as pairs of value and its modification. The format of such files is presented in detail in the following chapter.
  - has_header_line
    A flag which needs to be set to True if the first line in the look-up file corresponds to a header line (i.e. does not contain an attribute value), or to False if there is no header line in the look-up file.
  - unicode_encoding
    The Unicode encoding (a string name) used for encoding the values in the look-up file.

  Note that the position_function is not required by this corruptor method because complete values will be exchanged rather than single characters or sequences of characters only.

  An example (taken from the generate-data-english.py module) of how to initialise a categorical values corruptor is given below.

```
surname_misspell_corruptor = corruptor.CorruptCategoricalValue(\
            lookup_file_name = 'lookup-files/surname-misspell.csv',
            has_header_line = False,
            unicode_encoding = 'ascii')
```

- CorruptDataSet

  This final class provides a mechanism to specify how the different corruptor objects are to be applied and how the 'duplicate' records of a data set are to be generated by modifying the 'original' records that were created by the GenerateDataSet class.

  The following input arguments need to be given when a data set corruption object is initialised (constructed):

- `number_of_mod_records`
  The number of modified (corrupted) records that are to be generated. This will correspond to the number of 'duplicate' records that are generated.
- `number_of_org_records`
  The number of 'original' records that were generated by the `GenerateDataSet` class.
- `attribute_name_list`
  This argument is the list of attributes that are to be generated for each record, and the sequence of how they are to be written into the output file. The argument is a list containing the names of attributes, for which an attribute generation method has been initialised.

  This list needs to be the same list that was given when the `GenerateDataSet` class was initialised.
- `max_num_dup_per_rec`
  This argument sets the maximum number of 'duplicate' (modified / corrupted) records that can be generated for a single 'original' record. This must be a positive integer number.
- `num_dup_dist`
  This argument sets the probability distribution used to create the number of 'duplicate' records for one 'original' record. Possible distributions are: '`uniform`', '`poisson`', and '`zipf`'.
- `max_num_mod_per_attr`
  This argument sets the maximum number of modifications that can be applied on a single attribute value for a given record. This must be a positive integer number.
- `num_mod_per_rec`
  This argument sets the number of modifications that are applied to every 'original' record.
- `attr_mod_prob_dict`
  This dictionary contains probabilities (between 0.0 and 1.0) that determine how likely an attribute is selected for random modification (corruption). The keys of this dictionary are attribute names and values are probability values. Not all attributes need to be listed in this dictionary, only the ones onto which modifications are to be applied (i.e. attributes with a probability larger than 0.0). An example of such a dictionary will be shown in Chapter 5.

  Important: The sum of the probabilities given in this dictionary must be 1.0.
- `attr_mod_data_dict`
  This argument is a dictionary which must contain the same keys as the previous `attr_mod_prob_dict` dictionary. For each of the keys (i.e. each attribute that is to be modified), a list must be given that contains pairs of probability values and the corresponding corruptor objects (one of the corruptors described above) that are to be applied on this attribute. For each attribute listed, the sum of probabilities given in its list must sum to 1.0. An example of such a dictionary will be shown in Chapter 5.

A full example of how to initialise a `CorruptDataSet` object is provided in Chapter 5.

The `CorruptDataSet` class has one method only:

```
corrupt_records(rec_dict)
```

This method is used to corrupt the 'original' records given in the `rec_dict`, and it will add the 'duplicate' (modified / corrupted) records to this data structure and return all generated records ('original' and 'duplicate'). An example of how this method is used is also given in Chapter 5.

CHAPTER

FOUR

LOOK-UP FILE FORMATS

As was described in Chapter 3, various look-up files are required by the data generator modules `generator.py` and `corruptor.py`. The general structure of all these look-up files is that their content are values that are separated by commas, and therefore all look-up files are in the Comma Separated Values (CSV) text file format. The `basefunctions.py` module contains a function to read and a function to write CSV files (as was described in Chapter 3). In this chapter the detailed formats for the different look-up files required by the data generator are presented, and examples are shown to illustrate these file formats.

All look-up files can contain comment lines, which are lines that start with a '`#`' character. These lines will be skipped. Empty lines will also be skipped. It is also possible for look-up files to have a header-line (which might contain the name of the attributes in the file). Such a header line is assumed to be the first line in the file. All methods that require a look-up file to be read have an input argument `has_header_line` which can be set to `True` (if a look-up file does start with a header line) or `False` (if it does not).

The content of look-up files can also be encoded using any Unicode character-set encoding, and the input argument `unicode_encoding` of methods that require a look-up file to be read needs to be set to the correct Unicode encoding.

One general assumption that holds in several types of look-up files is that the counts of values (such as counts of categorical values) must be positive integer values. Examples to illustrate this will be given and discussed below.

4.1 Frequency look-up file format

Frequency look-up files are required by the class `GenerateFreqAttribute`, which can be used to generate categorical values where the likelihood that a value is generated depends upon the frequency (or count) of the value given in the look-up file used.

The structure of such frequency look-up files is very simple. Each row (or line) in the file must contain two values, the first being a categorical value and the second its frequency or count. Categorical values are assumed to be strings (even numbers are handled as strings) and they must not be empty. Each categorical value can only occur once in a frequency look-up file. Frequencies or counts must be positive integer values. A count of zero or a negative count will trigger an exception and cause the data generator program to stop, as will an empty categorical value.

An example frequency look-up file follows.

```
# Example selection of surnames and their frequencies/counts
anderson,14
baxter,42
miller,113
smith,213
zwillenberg,1
```

4.2 Categorical-categorical attribute look-up file format

The compound attribute type `GenerateCateCateCompoundAttribute` described in Chapter 3 requires a look-up file which contains the details of the categorical values of the two attributes that are generated and the respective counts for each of these value.

The format of such look-up files is as follows:

```
cate_attr1_val1, count, cate_attr2_val1, count1, cate_attr2_val2, count2, \
cate_attr2_val3, count3, cate_attr2_val4, count4, \
cate_attr2_val5, count5, cate_attr2_val6, count6
cate_attr1_val2, count, cate_attr2_val1, count1, cate_attr2_val2, count2, \
cate_attr2_val3, count3, cate_attr2_val4,count4
...
```

There are two different types of rows (lines) in such a look-up file.

- The first type of lines starts with a categorical value of the first attribute (`cate_attr1_val1` and `cate_attr1_val2` in the above example). Following this categorical value is its respective count (`count`, a positive integer number). The third column onwards contain pairs of categorical values of the second attribute and their respective positive counts (`cate_attr2_val1`, `count1`, `cate_attr2_val2`, `count2`, etc.)

  The last character (after a comma!) in a line of this type must be a backslash character (\). This signifies that the following line contains further categorical values of the second attribute (and their counts).

- The second type of lines does not contain a categorical value of the first attribute (and its count), but they are continuations of pairs of categorical values of the second attribute and their respective positive counts. If such a line ends with a backslash character (\) after a comma, then another such 'continuation' line will follow. If a line does not end with a backslash character, then it is assumed that the line following will be a line of the first type and that it contains the next categorical value of the first attribute.

The following small example (taken from the file '`gender-income.csv`' from the '`lookup-files`' folder) illustrates the format of look-up files for the `GenerateCateCateCompoundAttribute` compound attribute type.

```
male,60, canberra,7, darwin,5, \
sydney,30, melbourne,40, \
perth,18
female,40,canberra,10,sydney,40, \
melbourne,20,brisbane,15,hobart,5, \
perth,10
```

In this example, for the first attribute the count for the '`male`' gender is 60 while the count for '`female`' is 40. Therefore, when records are being generated, the likelihood that a gender value is randomly selected to be male is 60% while the likelihood that it is randomly selected to be female is 40%.

For a record where the gender has been set to male, with a 7% likelihood the city attribute will be randomly set to '`canberra`', with 5% likelihood it will be set to '`darwin`', with a 30% likelihood it will be set to '`sydney`', and so on. On the other hand, if the gender has been set to female for a certain record, then the likelihood that the city attribute will be selected to be '`canberra`' is 10%, the likelihood for it to be '`sydney`' is 10%, and so on.

As can be seen in this example, the different categorical values of the first attribute do not need to have the same categorical values for the second attribute listed (an overlap of values would generally be assumed though).

4.3 Categorical-continuous attribute look-up file format

The compound attribute type `GenerateCateContCompoundAttribute` described in Chapter 3 requires a look-up file which contains the details of the categorical values of the first attribute and for each of them the function and its parameters used for the continuous attribute.

The format of such look-up files is as follows:

```
cate_val1, count1, funct_name,funct_param_1,...,funct_param_N
cate_val2, count2, funct_name,funct_param_1,...,funct_param_N
cate_val3, count3, funct_name,funct_param_1,...,funct_param_N
...
```

Each line in such a look-up file contains the following information:

- The first element (first column) is a categorical value from the categorical attribute.
- The second element is its (positive) count, which determines the likelihood that the value will be randomly selected for a record during the data generation process.
- The third element is the name of the function that will be used to generate the continuous values of the second attribute for records where the categorical value given in this row of the look-up file has been randomly selected. Currently two functions are implemented: '`uniform`' and '`normal`'.
- If the function selected is the '`uniform`' function, then two parameters need to be given, which are `min_val` and `max_val`. These two parameters have the same meaning as the parameters given to the `generate_-uniform_value` function in the `attrgenfunct.py` module which was described in Chapter 3.

  If the function selected is the '`normal`' function, then four parameters need to be given, which are `mu`, `sigma`, `min_val`, and `max_val`. These four parameters have the same meaning as the parameters given to the `generate_normal_value` function in the `attrgenfunct.py` module which was described in Chapter 3. Both the `min_val` and `max_val` parameters can be set to '`None`' in which case no minimum and/or maximum limits will be enforced.

The following small example (taken from the file '`gender-income.csv`' from the '`lookup-files`' folder) illustrates the format of look-up files for the `GenerateCateContCompoundAttribute` compound attribute type.

```
male,30,uniform,20000,100000
female,40,normal,35000,100000,10000,None
unkown,30,normal,55000,45000,0,150000
```

In this example, for the categorical attribute the count for the '`male`' gender is 30, the count for '`female`' is 40, and the count for category '`unknown`' gender is 30. Therefore, when records are being generated, the likelihood that a gender value is randomly selected to be male is 30%, the likelihood that it is randomly selected to be female is 40%, and the likelihood that a gender value is randomly selected to be '`unknown`' is 30%.

For a record where the gender has been set to male, income values will be randomly generated between $20,000$ and $100,000$ with a uniform distribution. For records with a female gender, on the other hand, income values will be normally distributed with an average of $35,000$, a standard deviation of $100,000$, a minimum value of $10,000$, and no upper limit for income. For records where the gender has been set to an unknown value, the income will again be normally distributed with an average of $55,000$, a standard deviation of $45,000$, a minimum value of 0, and an upper limit of $150,000$.

4.4 Categorical-categorical-continuous attribute look-up file format

The compound attribute type `GenerateCateCateContCompoundAttribute` described in Chapter 3 requires a look-up file which contains the details of the categorical values of the first two attributes and for each combination of them the function and its parameters used for the continuous attribute.

The format of such look-up files is as follows:

```
cate_attr1_val1, count
cate_attr2_val1, count1, funct_name,funct_param_1,...,funct_param_N
cate_attr2_val2, count2, funct_name,funct_param_1,...,funct_param_N
cate_attr2_val3, count3, funct_name,funct_param_1,...,funct_param_N
...
cate_attr2_valX, countX,funct_name,funct_param_1,...,funct_param_N
cate_attr1_val2, count
cate_attr2_val1, count1,funct_name,funct_param_1,...,funct_param_N
cate_attr2_val2, count2,funct_name,funct_param_1,...,funct_param_N
cate_attr2_val3, count3,funct_name,funct_param_1,...,funct_param_N
...
cate_attr2_valX, countX,funct_name,funct_param_1,...,funct_param_N
cate_attr1_val3, count
...
```

There are two types of lines in such a look-up file. The first type contains a categorical value of the first categorical attribute and its positive count. Such a line is then followed by one or more lines that contain the following information:

- The first element (first column) is a categorical value from the second categorical attribute.
- The second element is the (positive) count of this categorical value, which determines the likelihood that the value will be randomly selected for the second categorical attribute in a record during the data generation process.
- The third element is the name of the function that will be used to generate the continuous values of the third attribute for records that have the categorical value given in this row of the look-up file in the second categorical attribute, and the categorical value for the first categorical attribute given in a previous line of the look-up file. Currently two functions are implemented: '`uniform`' and '`normal`'.
- If the function selected is the '`uniform`' function, then two parameters need to be given, which are `min_val` and `max_val`. These two parameters have the same meaning as the parameters given to the `generate_uniform_value` function in the `attrgenfunct.py` module which was described in Chapter 3.

  If the function selected is the '`normal`' function, then four parameters need to be given, which are `mu`, `sigma`, `min_val`, and `max_val`. These four parameters have the same meaning as the parameters given to the `generate_normal_value` function in the `attrgenfunct.py` module which was described in Chapter 3. Both the `min_val` and `max_val` parameters can be set to '`None`' in which case no minimum and/or maximum limits will be enforced.

The following small example (taken from the file '`gender-city-income.csv`' from the '`lookup-files`' folder) illustrates the format of look-up files for the `GenerateCateCateContCompoundAttribute` compound attribute type.

```
male,60
canberra,20,uniform,50000,90000
sydney,30,normal,75000,50000,20000,None
melbourne,30,uniform,35000,200000
perth,20,normal,55000,250000,15000,None
female,40
canberra,10,normal,45000,10000,None,150000
sydney,40,uniform,60000,200000
melbourne,20,uniform,50000,1750000
brisbane,30,normal,55000,20000,20000,100000
```

In this example, for the first categorical attribute the count for the '`male`' gender is 60 and the count for '`female`' is 40. When records are being generated, the likelihood that a gender value is randomly selected to be male is 60%, while the likelihood that it is randomly selected to be female is 40%.

For a record where the gender has been set to male, the possible city values are '`canberra`' with 20% likelihood, '`sydney`' with 30% likelihood, '`melbourne`' also with 30%likelihood, and '`perth`' with 20% likelihood. If the value randomly chosen for the second categorical value was '`canberra`', for example, then an income value is randomly chosen between $50,000$ and $90,000$ with a uniform distribution, while if the city selected was '`sydney`' then an income value is randomly chosen following a normal distribution with average value $75,000$ and standard deviation $50,000$, a minimum limit of $20,000$, and no upper limit.

4.5 Optical Character Recognition (OCR) variations look-up file format

The `CorruptValueOCR` corruptor class in the `corruptor.py` module requires a look-up file that contains character sequences and their OCR variations. These correspond to single or multiple characters that are replaced with another character (or other characters) mimicking OCR scanning errors.

The format of such look-up files is a simple two column (comma separated) format, where the first column contains a single character or a sequence of several characters, and the second column contains the OCR variation which can again be a single character or a sequence of several characters. It is assumed that these OCR modifications are symmetric, such that if a sequence occurring in the second column is found in an attribute value it can be replaced by the corresponding value in the first column.

The following small example shows a sub-set of all OCR modifications given in the file '`ocr-variations.csv`' as available in the folder '`lookup-files`':

```
5,S
5,s
2,Z
2,z
1,|
6,G
m,rn
cl,d
l>,b
```

4.6 Phonetic variations look-up file format

The `CorruptValuePhonetic` corruptor class in the `corruptor.py` module requires a look-up file that contains the patterns that govern where and how phonetic modification scan be applied. The details of this approach to phonetic modifications is provided in [5].

The format of such look-up files is a CSV file with seven columns that contain the following information:

1. Where a phonetic modification can be applied. Possible values are: 'ALL' (everywhere in a value), 'START', 'END', and 'MIDDLE'.
2. The original character or character sequence (i.e. the character(s) to be replaced).
3. The new character or character sequence which will replace the original character(s).
4. Pre-condition: A condition that must occur before the original string character sequence in order for this rule to become applicable. A value of 'None' means there is no pre-condition.
5. Post-condition: Similarly, a condition that must occur after the original string character sequence in order for this rule to become applicable. Again, a value of 'None' means there is no post-condition.
6. Pattern existence condition: This condition requires that a certain given string character sequence does ('y' flag) or does not ('n' flag) occur in the input string.
7. Start existence condition: Similarly, this condition requires that the input string starts with a certain string pattern ('y' flag) or not ('n' flag).

A special character used in such look-up files is '@'. It signifies an empty character or string.

The following small example shows a sub-set of all phonetic modifications given in the file 'phonetic-variations.csv' as available in the folder 'lookup-files':

```
ALL,h,@,None,None,None,None
END,e,@,None,None,None,None
ALL,t,d,None,None,None,None
ALL,d,t,None,None,None,None
ALL,c,k,None,None,None,None
ALL,w,@,None,None,None,None
ALL,nn,n,None,None,None,None
ALL,ll,l,None,None,None,None
```

The first rule for example corresponds to the removal of a 'h' character (replacing it with an empty string) anywhere in an attribute value, while the second rule corresponds to the removal of an 'e' only at the end of an attribute value.

4.7 Categorical value variations look-up file format

The CorruptCategoricalValue corruptor class in the corruptor.py module requires a look-up file that contains categorical values and their variations, such as their misspellings, nicknames, etc. This corruptor is suitable for attributes such as names (given names, surnames, town names, etc.) where there are either known spelling variations and nicknames (such as 'gail' and 'gayle', 'dickson' and 'dixon', or 'robert' and 'bob'), or known misspellings.

The format of such look-up files is a simple CSV file with two columns, where the first column contains categorical values and the second column contains their variations. For a given categorical value there can be several variations in the same look-up file, one per line (row) of the file as is shown in the example below.

The following small example shows a sub-set of all surname variations given in the file 'surname-misspell.csv' which is available in the folder 'lookup-files':

```
barclay,berkeley
baxter,bax
blythe,blithe
boleslaw,bolestlaw
bowden,bowen
boyle,oboyle
bree,obree
brice,bryce
brian,brien
brian,briant
brian,bryan
brian,bryant
brian,obrian
brian,obrien
brian,obryan
```

CHAPTER

FIVE

EXAMPLE DATA GENERATION MODULE

In this chapter, the module `generate-data-english.py` is described in detail to provide the user with the information necessary to understand how the data generator works, and how such a file can be modified.

A user can run such a program from terminal window (shell) by simply typing:

```
python generate-data-english.py
```

All parameters necessary to set a data generation process are defined within the `generate-data-english.py` module as will be described in the remainder of this chapter.

```
# =============================================================================
# generate-data-english.py - Python module to generate synthetic data based on
#                            English look-up and error tables.
#
# Peter Christen and Dinusha Vatsalan, January-March 2012
# =============================================================================
#
#  This Source Code Form is subject to the terms of the Mozilla Public
#  License, v. 2.0. If a copy of the MPL was not distributed with this
#  file, You can obtain one at http://mozilla.org/MPL/2.0/.
#
# =============================================================================
```

The program header contains information about the developers of this software and the license that covers the software. This header and licence notice must not be removed.

```
# Import the necessary other modules of the data generator
#
import basefunctions  # Helper functions
import attrgenfunct   # Functions to generate independent attribute values
import contdepfunct   # Functions to generate dependent continuous attribute
                      # values
import generator      # Main classes to generate records and the data set
import corruptor      # Main classes to corrupt attribute values and records
```

All five modules of the data generator system need to be imported so their functionalities (classes, methods and functions) are available to this program.

```
import random
random.seed(42)  # Set seed for random generator, so data generation can be
                 # repeated
```

Because much of the data generation (and modification) process is dependent on randomly selecting or calculating attribute values and choosing modification options, setting the Python random generator to specific initial state with the `random.seed` function will allow the repetition of a data generation process. Assuming the same random seed has been used, the same data set should be generated.

```
# Set the Unicode encoding for this data generation project. This needs to be
# changed to another encoding for different Unicode character sets.
# Valid encoding strings are listed here:
# http://docs.python.org/library/codecs.html#standard-encodings
#
unicode_encoding_used = 'ascii'
```

The data generator allows data in different Unicode encodings to be generated. This parameter, which can be set to any of the valid standard Unicode encodings listed at the given Web page, sets the Unicode encoding used both for look-up tables as well as for the encoding of the generated data that is to be written into the output file (see below). Chapter 8 covers the topic of Unicode encodings and required modifications to the data generator programs in more detail.

```
# The name of the record identifier attribute (unique value for each record).
# This name cannot be given as name to any other attribute that is generated.
#
rec_id_attr_name = 'rec-id'
```

Each generated record will be given a unique identifier of the form '`rec-`*X*`-org`' for 'original' records and '`rec-`*X*`-dup-`*Y*' for duplicate records, where *X* is the record number and *Y* is the duplicate number of the given original record. Both *X* and *Y* start from 0.

The `rec_id_attr_name` parameter sets the name of the record identifier attribute in the header line of the output data set, as can be seen in the small example data sets shown in Chapter 1.

```
# Set the file name of the data set to be generated (this will be a comma
# separated values, CSV, file).
#
out_file_name = 'example-data-english.csv'
```

This parameter sets the name of the output file that will be generated. The file will be of type Comma Separated Values (CSV).

```
# Set how many original and how many duplicate records are to be generated.
#
num_org_rec = 10000
num_dup_rec = 10000
```

These two parameters set how many 'original' and how many 'duplicate' records will be generated. The maximum number of 'duplicate' records that can be generated depends both upon the number of 'original' records and the parameter `max_duplicate_per_record` given next. Specifically, the number of 'duplicate' records cannot be larger than the number of 'original' records times the maximum number of 'duplicate' records that can be generated per 'original' record.

```
    # Set the maximum number of duplicate records can be generated per original
    # record.
    #
    max_duplicate_per_record = 3
```

This parameter sets the maximum number of 'duplicate' records that can be generated for a single 'original' record. This number must be a positive integer value.

```
    # Set the probability distribution used to create the duplicate records for one
    # original record (possible values are: 'uniform', 'poisson', 'zipf').
    #
    num_duplicates_distribution = 'zipf'
```

If more than one 'duplicate' record can be generated per 'original' record (i.e. if `max_duplicate_per_record` is set to a value larger than 1), then this parameter governs the overall distribution of how many 'duplicate' records will be generated per 'original' record (i.e. the likelihood that one 'original' record will have one, two or more 'duplicate' records generated for it).

```
    # Set the maximum number of modification that can be applied to a single
    # attribute (field).
    #
    max_modification_per_attr = 1
```

This parameter sets the maximum number of modifications that can be applied to a single attribute value in a certain record during the corruption process when 'duplicate' records are being created. This must be a positive integer number.

```
    # Set the number of modification that are to be applied to a record.
    #
    num_modification_per_record = 5
```

This parameter sets how many attributes will be modified in every 'duplicate' record. This must be a positive integer number.

```
    # Check if the given the unicode encoding selected is valid.
    #
    basefunctions.check_unicode_encoding_exists(unicode_encoding_used)
```

This code checks that the Unicode encoding provided is valid before the data generation objects are being initialised.

The following code sequence initialises the individual attribute generation objects based on the `GenerateFreqAttribute` and `GenerateFuncAttribute` classes that were described in Chapter 3.

```
# -----------------------------------------------------------------------------
# Define the attributes to be generated (using methods from the generator.py
# module).
#
gname_attr = \
    generator.GenerateFreqAttribute(attribute_name = 'given-name',
                          freq_file_name = 'lookup-files/givenname_f_freq.csv',
                          has_header_line = False,
                          unicode_encoding = unicode_encoding_used)

sname_attr = \
    generator.GenerateFreqAttribute(attribute_name = 'surname',
                          freq_file_name = 'lookup-files/surname-freq.csv',
                          has_header_line = False,
                          unicode_encoding = unicode_encoding_used)

postcode_attr = \
    generator.GenerateFreqAttribute(attribute_name = 'postcode',
                          freq_file_name = 'lookup-files/postcode_act_freq.csv',
                          has_header_line = False,
                          unicode_encoding = unicode_encoding_used)

phone_num_attr = \
    generator.GenerateFuncAttribute(attribute_name = 'telephone-number',
                       function = attrgenfunct.generate_phone_number_australia)

credit_card_attr =  \
    generator.GenerateFuncAttribute(attribute_name = 'credit-card-number',
                       function = attrgenfunct.generate_credit_card_number)

age_uniform_attr = \
    generator.GenerateFuncAttribute(attribute_name = 'age-uniform',
                       function = attrgenfunct.generate_uniform_age,
                       parameters = [0,120])

income_normal_attr = \
    generator.GenerateFuncAttribute(attribute_name = 'income-normal',
                       function = attrgenfunct.generate_normal_value,
                       parameters = [50000,20000, 0, 1000000, 'float2'])

rating_normal_attr = \
    generator.GenerateFuncAttribute(attribute_name = 'rating-normal',
                       function = attrgenfunct.generate_normal_value,
                       parameters = [0.0,1.0, None, None, 'float9'])
```

Each initialised attribute needs to be given a unique name (via the `attribute_name` argument). A user can modify these arguments by providing their own look-up files and attribute generation functions (which were described in Chapter 3).

Compound attributes are initialised next. Again, each attribute that will be generated needs to have a name that uniquely identifies it. Details about the arguments required for the different compound attribute types are given in Chapter 3, which the format of the different look-up tables used is described in the previous chapter.

```
gender_city_comp_attr = \
    generator.GenerateCateCateCompoundAttribute(\
          categorical1_attribute_name = 'gender',
          categorical2_attribute_name = 'city',
          lookup_file_name = 'lookup-files/gender-city.csv',
          has_header_line = True,
          unicode_encoding = 'ascii')

sex_income_comp_attr = \
    generator.GenerateCateContCompoundAttribute(\
          categorical_attribute_name = 'sex',
          continuous_attribute_name = 'income',
          continuous_value_type = 'float1',
          lookup_file_name = 'lookup-files/gender-income.csv',
          has_header_line = False,
          unicode_encoding = 'ascii')

gender_town_salary_comp_attr = \
    generator.GenerateCateCateContCompoundAttribute(\
          categorical1_attribute_name = 'alternative-gender',
          categorical2_attribute_name = 'town',
          continuous_attribute_name = 'salary',
          continuous_value_type = 'float4',
          lookup_file_name = 'lookup-files/gender-city-income.csv',
          has_header_line = False,
          unicode_encoding = 'ascii')

age_blood_pressure_comp_attr = \
    generator.GenerateContContCompoundAttribute(\
          continuous1_attribute_name = 'age',
          continuous2_attribute_name = 'blood-pressure',
          continuous1_funct_name =     'uniform',
          continuous1_funct_param =    [10,110],
          continuous2_function = contdepfunct.blood_pressure_depending_on_age,
          continuous1_value_type = 'int',
          continuous2_value_type = 'float3')

age_salary_comp_attr = \
    generator.GenerateContContCompoundAttribute(\
          continuous1_attribute_name = 'age2',
          continuous2_attribute_name = 'salary2',
          continuous1_funct_name =     'normal',
          continuous1_funct_param =    [45,20,15,130],
          continuous2_function = contdepfunct.salary_depending_on_age,
          continuous1_value_type = 'int',
          continuous2_value_type = 'float1')
```

In the following code block, different corruptor objects are initialised using the different classes available in the `corruptor.py` module that were described in Chapter 3.

Note that most of the corruptor methods are independent of the attribute they are applied on. They however depend upon the type of values that are expected in an attribute, and the types and distributions of modifications that one wants to apply for a certain attribute. For example, a corruptor of type `CorruptCategoricalValue`, which replaces known categorical values with variations of them, obviously depends upon the values in the categorical attribute and the values in the look-up file used. If these values (such as misspellings of known surname as in the given example) differ from the values given in this attribute in the 'original' records, then no modification of attribute values will occur.

```
# -----------------------------------------------------------------------------
# Define how the generated records are to be corrupted (using methods from
# the corruptor.py module).

# For the value edit corruptor, the sum or the four probabilities given must
# be 1.0.
#
edit_corruptor = \
    corruptor.CorruptValueEdit(\
          position_function = corruptor.position_mod_normal,
          char_set_funct = basefunctions.char_set_ascii,
          insert_prob = 0.5,
          delete_prob = 0.5,
          substitute_prob = 0.0,
          transpose_prob = 0.0)

edit_corruptor2 = \
    corruptor.CorruptValueEdit(\
          position_function = corruptor.position_mod_uniform,
          char_set_funct = basefunctions.char_set_ascii,
          insert_prob = 0.25,
          delete_prob = 0.25,
          substitute_prob = 0.25,
          transpose_prob = 0.25)

surname_misspell_corruptor = \
    corruptor.CorruptCategoricalValue(\
          lookup_file_name = 'lookup-files/surname-misspell.csv',
          has_header_line = False,
          unicode_encoding = unicode_encoding_used)

ocr_corruptor = corruptor.CorruptValueOCR(\
          position_function = corruptor.position_mod_normal,
          lookup_file_name = 'lookup-files/ocr-variations.csv',
          has_header_line = False,
          unicode_encoding = unicode_encoding_used)

keyboard_corruptor = corruptor.CorruptValueKeyboard(\
          position_function = corruptor.position_mod_normal,
          row_prob = 0.5,
          col_prob = 0.5)

phonetic_corruptor = corruptor.CorruptValuePhonetic(\
          lookup_file_name = 'lookup-files/phonetic-variations.csv',
          has_header_line = False,
          unicode_encoding = unicode_encoding_used)

missing_val_corruptor = corruptor.CorruptMissingValue()

postcode_missing_val_corruptor = corruptor.CorruptMissingValue(\
       missing_val='missing')

given_name_missing_val_corruptor = corruptor.CorruptMissingValue(\
       missing_value='unknown')
```

The list defined in the following code block determines which of the defined attribute generation objects will actually be generated, and the way they are to be written into the output file. The sequence in which attributes are listed will determine the order of how they are written as columns into the output file. Note that the first column of any generated output file will always be the record identifier attribute.

```
# -----------------------------------------------------------------------------
# Define the attributes to be generated for this data set, and the data set
# itself
#
attr_name_list = ['gender', 'given-name', 'surname', 'postcode', 'city',
                  'telephone-number', 'credit-card-number', 'income-normal',
                  'age-uniform', 'income', 'age', 'sex', 'blood-pressure']
```

The following list collects all the defined attribute generation objects so they can be easily passed to the data generation object below. Note that the order of this list is irrelevant, i.e. it does not determine how attributes are written into the output file. It is also possible to include attribute generation objects into this list that are not generated and written into the output file (this list therefore simply collects all attribute generation objects that have been initialised earlier on).

```
attr_data_list = [gname_attr, sname_attr, postcode_attr, phone_num_attr,
                  credit_card_attr, age_uniform_attr, income_normal_attr,
                  gender_city_comp_attr, sex_income_comp_attr,
                  gender_town_salary_comp_attr, age_blood_pressure_comp_attr,
                  age_salary_comp_attr]
```

In this following code the main data set generation object is initialised by passing all relevant initialised objects and parameters as input arguments. The requirements of these input arguments are described in Chapter 3.

The only argument the user might have to change in this initialisation is the setting of the header line for the output file to `True` (so a header line with the attribute names is written) or `False` (so no header line is written).

Note that the initialisation of this data set generation object does not yet mean that the records for this data set will be generated. The actual generation process is started later on in the program, as will be explained further below.

```
# Nothing to change here - set-up the data set generation object.
#
test_data_generator = generator.GenerateDataSet(output_file_name = \
                                          out_file_name,
                                          write_header_line = True,
                                          rec_id_attr_name = rec_id_attr_name,
                                          number_of_records = num_org_rec,
                                          attribute_name_list = attr_name_list,
                                          attribute_data_list = attr_data_list,
                                          unicode_encoding = \
                                                         unicode_encoding_used)
```

The following three code blocks contain the settings for the corruption process where ‘original’ records are modified into ‘duplicate’ records).

```
# Define the probability distribution of how likely an attribute will be
# selected for a modification.
# Each of the given probability values must be between 0 and 1, and the sum of
# them must be 1.0.
# If a probability is set to 0 for a certain attribute, then no modification
# will be applied on this attribute.
#
attr_mod_prob_dictionary = {'gender':0.1, 'given-name':0.2,'surname':0.2,
                            'postcode':0.1,'city':0.1, 'telephone-number':0.15,
                            'credit-card-number':0.1,'age':0.05}
```

The Python dictionary shown above defines how likely an attribute in the data set will be selected for the application of a modification into the attribute's value. The keys of this dictionary must be attribute names that are also listed in the `attr_name_list` defined before. The values of the dictionary are probability values (between 0.0 and 1.0) that set the likelihood of selection for a modification. The sum of these probabilities must be 1.0, otherwise the program will stop with an exception.

Attributes that are listed in the `attr_name_list` but that are not included in this above dictionary, and attributes that have their probability set to 0.0 will not be selected for modification. Therefore, the values in such attributes in all 'duplicate' records will be the same as the values in their corresponding 'original' records.

The Python dictionary shown below defines for each attribute where modifications are to be introduced (i.e. attributes that are listed in the above `attr_mod_prob_dictionary`) the actual corruptor methods that can be applied on an attribute and the probabilities that they are applied. The sum of probabilities given for each attribute has to be 1.0.

In the given dictionary, the only corruptor that will be applied on the '`gender`' attribute is a missing values corruptor which will replace a gender value with an empty string. On the other hand, for the '`surname`' attribute, modifications will be based on four different corruptors, the first (selected with a 10% likelihood) will replace a surname value with a known misspelling, the second (selected also with a 10% likelihood) will apply an OCR modification, the third (selected also with a 10% likelihood) will apply a keyboard modification, and the last (selected with a 70% likelihood) will modify a surname value based on a phonetic variation pattern.

```
# Define the actual corruption (modification) methods that will be applied on
# the different attributes.
# For each attribute, the sum of probabilities given must sum to 1.0.
#
attr_mod_data_dictionary = {'gender':[(1.0, missing_val_corruptor)],
                            'surname':[(0.1, surname_misspell_corruptor),
                                       (0.1, ocr_corruptor),
                                       (0.1, keyboard_corruptor),
                                       (0.7, phonetic_corruptor)],
                            'given-name':[(0.1, edit_corruptor2),
                                          (0.1, ocr_corruptor),
                                          (0.1, keyboard_corruptor),
                                          (0.7, phonetic_corruptor)],
                            'postcode':[(0.8, keyboard_corruptor),
                                        (0.2, postcode_missing_val_corruptor)],
                            'city':[(0.1, edit_corruptor),
                                    (0.1, missing_val_corruptor),
                                    (0.4, keyboard_corruptor),
                                    (0.4, phonetic_corruptor)],
                            'age':[(1.0, edit_corruptor2)],
                            'telephone-number':[(1.0, missing_val_corruptor)],
                            'credit-card-number':[(1.0, edit_corruptor)]}
```

In this following code the main data set corruption object is initialised by passing all relevant initialised objects and parameters as input arguments. The requirements of these input arguments are described in Chapter 3. Nothing needs to be changed here by a user.

```
# Nothing to change here - set-up the data set corruption object
#
test_data_corruptor = corruptor.CorruptDataSet(number_of_org_records = \
                                          num_org_rec,
                                          number_of_mod_records = num_dup_rec,
                                          attribute_name_list = attr_name_list,
                                          max_num_dup_per_rec = \
                                                 max_duplicate_per_record,
                                          num_dup_dist = \
                                                 num_duplicates_distribution,
                                          max_num_mod_per_attr = \
                                                 max_modification_per_attr,
                                          num_mod_per_rec = \
                                                 num_modification_per_record,
                                          attr_mod_prob_dict = \
                                                 attr_mod_prob_dictionary,
                                          attr_mod_data_dict = \
                                                 attr_mod_data_dictionary)
```

The final block of code shown below now starts the actual data generation and corruption process. First, the selected number of ‘original’ records is generated by the call to the `test_data_generator.generate()` method. In a second step, the selected number of ‘duplicated’ (modified / corrupted) records is generated and added to the dictionary of all records (`rec_dict`). Finally, the whole set of generated records is written into the output file.

```
# =============================================================================
# No need to change anything below here

# Start the data generation process
#
rec_dict = test_data_generator.generate()

assert len(rec_dict) == num_org_rec  # Check the number of generated records

# Corrupt (modify) the original records into duplicate records
#
rec_dict = test_data_corruptor.corrupt_records(rec_dict)

assert len(rec_dict) == num_org_rec+num_dup_rec # Check total number of records

# Write generate data into a file
#
test_data_generator.write()

# End.
# =============================================================================
```

CHAPTER

SIX

INSTALLATION

The complete data generator is supplied as a compressed file archive in both the ZIP and TAR.GZ formats to facilitate easy installation of the generator on computer systems running different operating systems. While the data generator has been developed and tested using Linux based (Ubuntu) computers with Python versions 6.2.5 and 7.2.2+, all programs have been developed to be independent of the operating system used. However, errors might occur, in which case the developers should be contacted with details of the problems encountered so they can be fixed.

The installation of this software requires the simple uncompression and extraction of the provided file archive. This process will generate a folder named:

```
anu-flab-data-generator-30march2012/
```

This folder contain the main modules of the data generator that have been described in previous chapters:

```
attrgenfunct.py
basefunctions.py
contdepfunct.py
corruptor.py
generator.py
generate-data-english.py
```

It also contains a copy of the **Mozilla Public License Version 2** under which this software is licensed

```
MPL2.0.txt
```

There are several folders in the main `anu-flab-data-generator-30march2012/` folder:

```
lookup-files/
tests/
docu/
papers/
```

The look-up files required by the data generator are stored in the `lookup-files/` folder. Their format has been described in Chapter 4. All testing programs and accompanying files are available in the folder `tests/`. Testing procedures are described in the following chapter. This documentation is available as PDF file in the `docu/` folder, and its source code (as Latex files) can be found in the folder `docu/source`. Finally, the folder `papers/` contains several publications that are relevant to this data generator.

CHAPTER

SEVEN

TESTING

This chapter covers the testing process used in the development of the flexible data generator. The five modules, basefunctions, attrgenfunct.py, contdepfunct.py, generator.py, and corruptor.py, are tested individually with one corresponding test program each. An integration testing of the whole data generator system is tested with several test cases through the mainTest.py program. The test scope includes the following:

1. Unit testing: Testing of all arguments and the functionality of all major functions and methods of each module.
2. Integration testing: End-to-end testing and testing of the integration of all modules that interact within the data generator system.

7.1 Testing programs

The testing programs are also written in the Python programming language. A Python standard module called unittest is used for the unit testing of the modules. One testing program for each module has been developed and all the testing programs reside in the tests/ folder. The testing program files are named as *module_name*Test.py. For example, the testing program for the basefunctions.py module is available in the basefunctionsTest.py file in the tests/ folder. The main testing of the functionality of the data generator system is available in the mainTest.py file which is also available in the tests/ folder.

To run the testing programs, the following commands need to be used in the Python interpretor started in the tests/ folder of the main data generator folder:

```
>>> execfile("basefunctionsTest.py")
>>> execfile("attrgenfunctTest.py")
>>> execfile("contdepfunctTest.py")
>>> execfile("generatorTest.py")
>>> execfile("corruptorTest.py")
>>> execfile("mainTest.py")
```

Alternatively, the test programs can be started directly from a terminal (assuming the location is set to the tests/ folder:

```
> python basefunctionsTest.py
> python attrgenfunctTest.py
> python contdepfunctTest.py
> python generatorTest.py
> python corruptorTest.py
> python mainTest.py
```

The unit testing of the modules tests the input arguments of each function and of each constructor of the classes in a module, as well as the functionality of each function and method in the modules.

Several test cases are used for testing both valid input arguments, known as 'normal' test type, and exception input arguments, known as 'exception' test type. The exception test type includes test cases of wrong values, wrong types, and missing values. Test results are checked against the expected output and reported in a log file in the `logs/` folder within the `tests/` folder. Details about the format of such log files are given below.

Functionality tests are written for each function to make sure that the function works correctly and returns the expected outcome. These results are also reported in the log file along with arguments testing output.

The `mainTest.py` program tests the overall functionality of the data generator system with several test cases. This module provides stress testing for the system by generating data with a large variety of parameter settings and checking that the data are generated according to the parameter settings.

Since many of the functions make use of the `random` function to generate random values, the command `random.seed(42)` is used when testing to make sure that the test results will be the same each time a testing program is run.

7.2 Testing log files

The log files are named as *module_name*`Test-`*date*`-`*time*`.csv`. These log files will contain the test results in comma separated values. The values that will be written into the log file are module name, class name, function name, input argument name, test type ('normal' or 'exception'), the number of patterns (test cases) that were tested, a summary of the testing (how many patterns passed and how many failed), and a description of failure (if any). If a module does not contain any classes and contains only functions, then the class name will be set to '`n/a`'. For the function testing, the value for input argument name will also be written as '`n/a`' and test type value will be '`funct`'.

An example of the test output format is illustrated in the following table:

Test results generated by basefunctionsTest.py
Test started: 20120329-10:30

Module name	Class name	Function name	Argument name	Test type	Patterns tested	Summary	Failure description
basefunctions	n/a	check_is_flag	variable	Normal	3	all tests passed	
basefunctions	n/a	check_is_flag	variable	Exception	6	all tests passed	
basefunctions	n/a	check_is_flag	value	Normal	6	all tests passed	
basefunctions	n/a	check_is_flag	value	Exception	6	all tests passed	
basefunctions	n/a	char_set_ascii	string_variable	Normal	5	1 test failed	failed test for input [123]
basefunctions	n/a	char_set_ascii	string_variable	Exception	6	all tests passed	

Test results generated by generatorTest.py
Test started: 20120329-12:30

Module name	Class name	Function name	Argument name	Test type	Patterns tested	Summary	Failure description
generator	GenerateFuncAttribute	constructor(__init__)	attribute_name	Normal	6	all tests passed	
generator	GenerateFuncAttribute	constructor(__init__)	attribute_name	Exception	6	1 test failed	failed test for input ['attr1']
generator	GenerateFuncAttribute	constructor(__init__)	function	Normal	6	all tests passed	
generator	GenerateFuncAttribute	constructor(__init__)	function	Exception	3	all tests passed	
generator	GenerateFuncAttribute	constructor(__init__)	parameters	Normal	6	all tests passed	
generator	GenerateFuncAttribute	constructor(__init__)	parameters	Exception	4	all tests passed	
generator	GenerateFuncAttribute	create_attribute_value	n/a	funct	10000	all tests passed	

The output of the `mainTest.py` program is a text file with a summary of the tests conducted and a description of tests if any of them failed. An example output follows:

```
Test results generated by mainTest.py
Test started: 20120329-1522


Test case parameters:
  rec_id_attr_name = rec_id
  num_org_rec = 100
  num_dup_rec = 100
  max_duplicate_per_record = 1
  num_duplicates_distribution = uniform
  max_modification_per_attr = 1
  num_modification_per_record = 1

  Distribution of duplicates: ("uniform" expected)
    1: 100 records

  All tests passed


Test case parameters:
  rec_id_attr_name = rec_id
  num_org_rec = 100
  num_dup_rec = 100
  max_duplicate_per_record = 1
  num_duplicates_distribution = poisson
  max_modification_per_attr = 1
  num_modification_per_record = 1

  Distribution of duplicates: ("poisson" expected)
    1: 100 records

  All tests passed
```

CHAPTER

EIGHT

UNICODE ADJUSTMENTS

The data generator was designed and implemented in such as way that it can be used for the generation of data sets encoded using different Unicode encodings with minimum modifications and adjustments needed by a user. This chapter describes which functions and methods might need to be modified.

Information about Unicode encoding functionalities within Python can for example be found at:

http://docs.python.org/howto/unicode.html

The following parameters, functions, modules and look-up files might need to be modified to facilitate the generation of data in different Unicode encodings:

- `unicode_encoding_used`

  This main parameter, which can be found at the beginning of the main modules `generate-data-english.py` and `generate-data-japanese.py`, controls which Unicode encoding is used for all look-up files that are used within the data generator, as well as the encoding that is used to write the generated data into the output file. This parameter has to be a string that corresponds to one of the many valid Python standard encodings that are listed at:

  http://docs.python.org/library/codecs.html#standard-encodings

- `get_char_ascii`

  The `basefunctions.py` module contains a function `get_char_ascii` which is used by the class `CorruptValueEdit` in the `corruptor.py` module. This function analyses a given input string and it returns a string that contains characters of the same type. For example, if an input string only contains letters, this function will return the string '`abcdefghijklmnopqrstuvwxyz`', while if the input string contains digits it will return the string '`0123456789`. The returned string contains all possible characters that can be used by the `CorruptValueEdit` when an insert or a substitution edit is being applied to an attribute value.

  For Unicode encodings different from the standard ASCII encoding, a user needs to implement a function that provides the same functionality as `get_char_ascii`, but that returns a string that contains all characters of a certain type. For example, for a Japanese function such as `get_char_kanji` or `get_char_hiragana` would analyse an input string to check if the input string does contain Kanji or Hiragana characters, and then return a string that contains all possible Kanji or Hiragana characters.

- Look-up files in the `lookup-files` folder

  Look-up files with categorical values in a given Unicode encoding are of course necessary to facilitate the generation of data in that given encoding schema. Several small example files are provided with this software:
  `gender-city-japanese.csv`
  `surname-freq-japanese.csv`
  `surname-misspell-japanese.csv`
  These files are used in the example module `generate-data-japanese.py` to generate a small data set containing Japanese surname and city name values.

- `attrgenfunct.py`

  The functions in this module generate values which are used in the data generation process to populate the generated records with values. For categorical (i.e. non-numerical) values that are generated by a function in this module the character set of the strings generated must be in the same Unicode encoding as the encoding selected in the main module through the parameter `unicode_encoding_used`, as otherwise encoding conflicts are likely to occur that potentially will lead to data that cannot be correctly recognised in a given Unicode encoding.

Note that due to limited access to different Unicode data files the developers of this software were only able to conduct limited testing of this software with different Unicode encodings. Therefore, any bugs or problems that are related to Unicode encoding are encountered they can be reported to the developers.

APPENDIX

A

MOZILLA PUBLIC LICENSE VERSION 2.0

All Python modules of the flexible data generator described in this document are covered my the **Mozilla Public License Version 2.0**, which is available from http://www.mozilla.org/MPL/2.0/, and included in the following.

1.7. "Larger Work"
means a work that combines Covered Software with other material, in a separate file or files, that is not Covered Software.

1.8. "License"
means this document.

1.9. "Licensable"
means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently, any and all of the rights conveyed by this License.

1.10. "Modifications"
means any of the following:

(a) any file in Source Code Form that results from an addition to, deletion from, or modification of the contents of Covered Software; or

(b) any new file in Source Code Form that contains any Covered Software.

1.11. "Patent Claims" of a Contributor
means any patent claim(s), including without limitation, method, process, and apparatus claims, in any patent Licensable by such Contributor that would be infringed, but for the grant of the License, by the making, using, selling, offering for sale, having made, import, or transfer of either its Contributions or its Contributor Version.

1.12. "Secondary License"
means either the GNU General Public License, Version 2.0, the GNU Lesser General Public License, Version 2.1, the GNU Affero General Public License, Version 3.0, or any later versions of those licenses.

1.13. "Source Code Form"
means the form of the work preferred for making modifications.

1.14. "You" (or "Your")
means an individual or a legal entity exercising rights under this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

```
2. License Grants and Conditions
--------------------------------

2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free,
non-exclusive license:

(a) under intellectual property rights (other than patent or trademark)
    Licensable by such Contributor to use, reproduce, make available,
    modify, display, perform, distribute, and otherwise exploit its
    Contributions, either on an unmodified basis, with Modifications, or
    as part of a Larger Work; and

(b) under Patent Claims of such Contributor to make, use, sell, offer
    for sale, have made, import, and otherwise transfer either its
    Contributions or its Contributor Version.

2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution
become effective for each Contribution on the date the Contributor first
distributes such Contribution.

2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under
this License. No additional rights or licenses will be implied from the
distribution or licensing of Covered Software under this License.
Notwithstanding Section 2.1(b) above, no patent license is granted by a
Contributor:

(a) for any code that a Contributor has removed from Covered Software;
    or

(b) for infringements caused by: (i) Your and any other third party’s
    modifications of Covered Software, or (ii) the combination of its
    Contributions with other software (except as part of its Contributor
    Version); or

(c) under Patent Claims infringed by Covered Software in the absence of
    its Contributions.

This License does not grant any rights in the trademarks, service marks,
or logos of any Contributor (except as may be necessary to comply with
the notice requirements in Section 3.4).

2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to
distribute the Covered Software under a subsequent version of this
License (see Section 10.2) or under the terms of a Secondary License (if
permitted under the terms of Section 3.3).

2.5. Representation

Each Contributor represents that the Contributor believes its
Contributions are its original creation(s) or it has sufficient rights
to grant the rights to its Contributions conveyed by this License.
```

```
2.6. Fair Use

This License is not intended to limit any rights You have under
applicable copyright doctrines of fair use, fair dealing, or other
equivalents.

2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted
in Section 2.1.

3. Responsibilities
-------------------

3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any
Modifications that You create or to which You contribute, must be under
the terms of this License. You must inform recipients that the Source
Code Form of the Covered Software is governed by the terms of this
License, and how they can obtain a copy of this License. You may not
attempt to alter or restrict the recipients' rights in the Source Code
Form.

3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

(a) such Covered Software must also be made available in Source Code
    Form, as described in Section 3.1, and You must inform recipients of
    the Executable Form how they can obtain a copy of such Source Code
    Form by reasonable means in a timely manner, at a charge no more
    than the cost of distribution to the recipient; and

(b) You may distribute such Executable Form under the terms of this
    License, or sublicense it under different terms, provided that the
    license for the Executable Form does not attempt to limit or alter
    the recipients' rights in the Source Code Form under this License.

3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice,
provided that You also comply with the requirements of this License for
the Covered Software. If the Larger Work is a combination of Covered
Software with a work governed by one or more Secondary Licenses, and the
Covered Software is not Incompatible With Secondary Licenses, this
License permits You to additionally distribute such Covered Software
under the terms of such Secondary License(s), so that the recipient of
the Larger Work may, at their option, further distribute the Covered
Software under the terms of either this License or such Secondary
License(s).

3.4. Notices

You may not remove or alter the substance of any license notices
(including copyright notices, patent notices, disclaimers of warranty,
or limitations of liability) contained within the Source Code Form of
the Covered Software, except that You may alter any license notices to
the extent required to remedy known factual inaccuracies.
```

```
************************************************************************
*                                                                      *
*  6. Disclaimer of Warranty                                           *
*  -------------------------                                           *
*                                                                      *
*  Covered Software is provided under this License on an "as is"       *
*  basis, without warranty of any kind, either expressed, implied, or  *
*  statutory, including, without limitation, warranties that the       *
*  Covered Software is free of defects, merchantable, fit for a        *
*  particular purpose or non-infringing. The entire risk as to the     *
*  quality and performance of the Covered Software is with You.        *
*  Should any Covered Software prove defective in any respect, You     *
*  (not any Contributor) assume the cost of any necessary servicing,   *
*  repair, or correction. This disclaimer of warranty constitutes an   *
*  essential part of this License. No use of any Covered Software is   *
*  authorized under this License except under this disclaimer.         *
*                                                                      *
************************************************************************

************************************************************************
*                                                                      *
*  7. Limitation of Liability                                          *
*  --------------------------                                          *
*                                                                      *
*  Under no circumstances and under no legal theory, whether tort      *
*  (including negligence), contract, or otherwise, shall any           *
*  Contributor, or anyone who distributes Covered Software as          *
*  permitted above, be liable to You for any direct, indirect,         *
*  special, incidental, or consequential damages of any character      *
*  including, without limitation, damages for lost profits, loss of    *
*  goodwill, work stoppage, computer failure or malfunction, or any    *
*  and all other commercial damages or losses, even if such party      *
*  shall have been informed of the possibility of such damages. This   *
*  limitation of liability shall not apply to liability for death or   *
*  personal injury resulting from such party’s negligence to the       *
*  extent applicable law prohibits such limitation. Some               *
*  jurisdictions do not allow the exclusion or limitation of           *
*  incidental or consequential damages, so this exclusion and          *
*  limitation may not apply to You.                                    *
*                                                                      *
************************************************************************

8. Litigation
-------------

Any litigation relating to this License may be brought only in the
courts of a jurisdiction where the defendant maintains its principal
place of business and such litigation shall be governed by laws of that
jurisdiction, without reference to its conflict-of-law provisions.
Nothing in this Section shall prevent a party’s ability to bring
cross-claims or counter-claims.
```

```
Exhibit B - "Incompatible With Secondary Licenses" Notice
---------------------------------------------------------

  This Source Code Form is "Incompatible With Secondary Licenses", as
  defined by the Mozilla Public License, v. 2.0.
```

BIBLIOGRAPHY

[1] P. Christen. Probabilistic data generation for deduplication and data linkage. In *IDEAL, Springer LNCS*, volume 3578, pages 109–116, Brisbane, 2005.

[2] P. Christen. Privacy-preserving data linkage and geocoding: Current approaches and research directions. In *Workshop on Privacy Aspects of Data Mining, held at IEEE ICDM*, Hong Kong, 2006.

[3] P. Christen. Development and user experiences of an open source data cleaning, deduplication and record linkage system. *SIGKDD Explorations*, 11(1):39–48, 2009.

[4] P. Christen, T. Churches, and M. Hegland. Febrl – A parallel open source data linkage system. In *PAKDD, Springer LNAI*, volume 3056, pages 638–647, Sydney, 2004.

[5] P. Christen and A. Pudjijono. Accurate synthetic generation of realistic personal information. In *PAKDD, Springer LNAI*, volume 5476, pages 507–514, Bangkok, Thailand, 2009.