# Study Unit 4 Data Management

## Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the operations on datasets
2. Prepare data for analysis using Python programming

## Overview

This unit will introduce the key data structure for analytics in Python: the pandas DataFrame. We will learn to develop Python programs to import data from external sources and convert them to DataFrames, and then to index and query these structures. We will then deepen our understanding of the pandas package by learning its efficient functionality on merging multiple DataFrames, identifying and dealing with missing data and outliers, sorting, grouping and transforming data, as well as discretising numeric variables to bins.

## Chapter 1 Import Data

Lesson Recording - Import Data in pandas

In the previous study units, we have learned the basic techniques of Python programming. In this and the next study units, we will discuss in detail how Python can be used for data management and data analytics.

The most common package for data management in Python is “pandas”. After installing pandas using pip, we can import it in our program by the following syntax:

|  |
| --- |
| import pandas as pd |

Here, we use the alias pd to refer to the pandas package in our programs.

To start working with pandas, we need to have Python compatible datasets. Data circulating in organisations or on the internet are mostly saved as text files or worksheets. Text editors, spreadsheets, and data management apps are popular tools for opening and working with them. Pandas actually provides the same possibilities. The first step here is to load a dataset in the Python environment and open it in the format of pandas. Suppose we have a dataset from an external source saved as a .csv text file; we can import it by the following pandas function:

|  |
| --- |
| DataFrame\_name = pd.read\_csv("csv\_file\_name.csv") |

The content stored in the file “csv\_file\_name.csv” will be then assigned to the pandas dataset object, or DataFrame, named DataFrame\_name. The function read\_csv() is called a reader since it reads in specific format of data files and converts them to pandas DataFrame.

Same as functions in NumPy or matplotlib, the read\_csv() function has more arguments than we list out here. We can adjust the execution of the reader to the specifications of the .csv file with these arguments. For instance, we can specify the character string of the delimiter, the row number in which the header is stored, the path of the .csv file, etc. You can refer to <https://pandas.pydata.org/docs/user_guide/‌io.html#io-read-csv-table> for further details.

Since .csv is not the only common file format of data files, pandas also allows the import of other file formats such as Excel spreadsheets, SPSS data or Stata data into Python by providing the functions listed in the following table.

Table 4.1 Most Common Data File Formats and the Corresponding Reader in pandas

|  |  |  |
| --- | --- | --- |
| **Reader** | **Format Type** | **Data Description** |
| read\_csv() | text | CSV |
| read\_html() | text | HTML |
| read\_clipboard() | text | Local clipboard |
| read\_excel() | binary | MS Excel |
| read\_stata() | binary | Stata |
| read\_sas() | binary | SAS |
| read\_spss() | binary | SPSS |
| read\_pickle() | binary | Python Pickle Format |
| read\_sql() | SQL | SQL |
| read\_gbq() | SQL | Google BigQuery |

We can then use the .head() method to display the first five rows of the imported dataset. It is important to check whether the data have been accurately imported.

|  |
| --- |
| DataFrame\_name.head() |

Alternatively, we can also use the conventional print() function to display the whole DataFrame. Nevertheless, this can be quite frustrating if the dataset contains many rows and columns, and the output does not fit to the window properly. Another way to print the whole DataFrame is to use the display() function or omit the function completely and simply execute a syntax with only the name of the DataFrame.

**Note:** When using NumPy ndarrays (n-dimensional arrays) to store multidimensional data, a burden is placed on the programmer to specify the orientation of the dataset, because axes are considered more or less equivalent. The meanings of rows and columns of an array do not differ significantly and switching the roles of rows and columns (transpose) to store the data does not change the nature of the array at all. And the functionality of the NumPy functions will remain. For pandas DataFrame, or datasets in general, the rows record individual observations, and the columns represent the features, or variables, of the data. Their roles usually do not change throughout the entire analysis process. Thus, the axes lend more semantic meaning to the data, and hence reduce the amount of mental effort required to code up data transformation.

|  |
| --- |
| **Example (Adult Census Data):** The US Adult Census dataset is a repository of 48,842 entries extracted from the 1994 US Census database. The dataset is downloadable from [https://www.kaggle.com/wenruliu/adult-income-dataset](https://www.kaggle.com/wenruliu/adult-income-datasetI). It is used to predict whether income would exceed $50,000 per year according to the 14 social-demographic attributes (Source: <http://www.cs.toronto.edu/~delve/data/‌adult/adultDetail.html)>. Below is a list of the available variables in the dataset:   1. **age**: the age of an individual. Its value can be any integer greater than 0. 2. **workclass**: a general term to represent the employment status of an individual. Its value can be Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked. 3. **fnlwgt**: final weight. In other words, this is the number of people the entry represents. Its value can be any integer greater than 0. 4. **education**: the highest level of education achieved by an individual. Its value can be Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool. 5. **educational-num**: the highest level of education achieved in numerical form. Its value can be any Integer greater than 0. 6. **marital-status**: marital status of an individual. Married-civ-spouse corresponds to a civilian spouse while Married-AF-spouse is a spouse in the Armed Forces. Its value can be Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse. 7. **occupation**: the general type of occupation of an individual. Its value can be Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces. 8. **relationship**: represents what this individual is relative to others. For example, an individual could be a husband. Each entry only has one relationship attribute. Its value can be Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried. 9. **race**: Descriptions of an individual’s race. Its value can be White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black. 10. **gender**: the biological gender of the individual. Its value can be Male, Female. 11. **capital-gain**: capital gains for an individual. Its value can be any integer greater than or equal to 0. 12. **capital-loss**: capital loss for an individual. Its value can be any integer greater than or equal to 0. 13. **hours-per-week**: the hours an individual has reported to work per week. Its value can be any positive real number or 0. 14. **native-country**: country of origin for an individual. Its value can be United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands. 15. **income**: whether or not an individual makes more than $50,000 annually. Its value can be <=50k, >50k.   Since the original data file is saved in .csv format, we can use the read\_csv() function to import the dataset to Python.    Figure 4.1 Importing Data with pandas  If we printed the census dataset using the print() function, the output would not fit the window at all.    Figure 4.2 Printing Entire Imported Dataset  As shown in Figure 4.2, the DataFrame will be truncated anyway. Eventually, only the head and the tail of the dataset will be displayed due to lack of space. As a result, it is more advisable to use the pd.head() than print() for control purpose.  Furthermore, Python provides the display() function.    Figure 4.3 Printing the Head and Tail of the Imported Dataset by display()  The display() function prints out the first 5 and last 5 observations of the DataFrame. The font is smaller than the one used by the print() function, and we are therefore able to see all columns side by side and without linebreak. Furthermore, it also shows the number of rows and columns in the DataFrame, which can be quite helpful in some cases. |

**Read**

Refer to the link below for more details and examples on the read\_csv() function of the pandas package:

<https://pandas.pydata.org/docs/user_guide/io.html#io-read-csv-table>

Read the following website for more information regarding the US Census data, including the explanation of the variable names and other useful information about the data:

[http://www.cs.toronto.edu/~delve/data/adult/ adultDetail.html)](http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html)

## Chapter 2 Data Selection

Lesson Recording - Data Selection in pandas

Same as Python lists or NumPy arrays, we can access a pandas DataFrame by using the index operator []. In this chapter, we will introduce three ways to subset rows, columns, or elements of a DataFrame.

### 2.1 Selecting Columns by Variable Names

To select specific columns, which represent the variables of a dataset, we can create a list with the variable names (or labels) to be selected and then put it in the index operator.

|  |
| --- |
| DataFrame\_name[["var\_name1", "var\_name2", …]] |

Note that each variable name must be put within a pair of quotation marks since it is treated as a string in this case. If we simply want to access one column, we can omit the creation of the list and put the variable name as string inside the index operator directly.

|  |
| --- |
| **Example (Cont’d):** Now we would like to select the columns "marital-status", "race", "gender" and "income" from the US Census dataset.    Figure 4.4 Select Columns from a DataFrame by Variable Names  Suppose we would like to select the target variable "income" alone and save the column as a NumPy array for further calculation.    Figure 4.5 Select a Single Column from a DataFrame and Save it as NumPy Array  We can see that no list will be needed within the index operator if only a single column is selected. After selecting the column, the resulting subset of the pandas DataFrame will then be converted to an NumPy array by the np.array() function. |

### 2.2 Selecting Rows by Positions and Indices

Accessing rows requires different techniques than accessing columns. While we can use the labels of the columns, or variable names, to select the columns we want, there are usually no natural “observation names” that we can refer to when selecting rows from a DataFrame. However, in Figure 4.1, we can see that a row index is provided at the beginning of every row by pandas. It starts with 0 and ends with the number of rows in the DataFrame minus one. As a result, rows can be queried by the numeric index position, starting at 0, using the DataFrame attribute iloc.

|  |
| --- |
| DataFrame\_name.iloc[start:end] |

The indices in the index operator do not need to be consecutive integers. It can be any integers within the range 0 and number of rows in the DataFrame – 1. But these integers must be put in a list first if there are more than one of them. If we want to select a single row instead, we can simply put one index in the index operator.

|  |
| --- |
| **Example (Cont’d):** Sometimes we may need to inspect a dataset after running some programs to adjust it for further analytics purposes. We have been introduced to the .head() method to print out the first five rows of the dataset. We can also randomly picked a few rows from the DataFrame for our inspection. Here, we need NumPy to draw random indices to select the rows for us.    Figure 4.6 Select Rows from a DataFrame by Random Row Indices  In the first line of our program, we determine the total number of rows in the dataset census. Note that the method .shape() is also applicable to pandas DataFrames and it returns a tuple (Total Row Number, Total Column Number) to us. As a result, we can refer to the first element of the tuple as the total number of rows in census. In the second line, we draw a total of 10 random integers from the interval 0 and nrow, the number of rows in census. Note that nrow as the upper boundary is not included in the drawing process at all. The integers drawn by the random.randint() function will then be assigned to the object named randrow. And these will then be used as the list of rows that we select from census. |

Before we can select rows based on their index labels, we need to create the index labels first by the method .set\_index().

|  |
| --- |
| DataFrame\_name.set\_index(key, inplace = True) |

The parameter key can be either a single variable name (column label), a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of variable names and arrays. The argument inplace controls whether the DataFrame should be modified in place or a new DataFrame should be created. If it is True, the changes will take place in the original DataFrame.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to group the US Census data by the occupation of the observations.    Figure 4.7 Setting the Values of a Column as Row Index  If we use one of the columns as the row index of our DataFrame, that column will not be a regular part of the dataset anymore. As you can see from Figure 4.7, the variable occupation has disappeared from the dataset.  Suppose we would like to change our row index from the occupation to the age group of each observation. In the first step, we need to remove the occupation as our row index by the .reset\_index() method. It reverses the effect of the .set\_index() method and removes the current row index and converts it back to a column in the DataFrame.    Figure 4.8 Resetting Row Index of a DataFrame  Then we recode the age in each row to our target grouping and store it in a new array. For observations less than 30 years, the group will be labelled as “Age <30”, whereas the label of those between 30 and 59 years old is “Age 30-59” and the rest receives the label “Age 60+”. Subsequently, we assign this array as the new row index.    Figure 4.9 Setting a NumPy Array as Row Index  We first create a list named agegroup that contains the grouping of each observation’s age. The list will then be converted into a NumPy array and used as the row index subsequently. |

The DataFrame rows can be queried by the row index labels using the .loc attribute.

|  |
| --- |
| DataFrame\_name.loc[["row\_label1", "row\_label2", …]] |

We can see that selecting rows from a DataFrame by the .loc attribute works in a very similar fashion as the column selection. The row labels must be indicated as strings and put in a list if we want to select more than one of them. If we just want to select rows of a single label, we can put the label as a string in the .loc attribute directly.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to select all the observations that are 30 years of age or younger from the US Census data.    Figure 4.10 Selecting Rows by Single Row Label  And if we want to select the youngest and oldest age groups from census, we will need to put the row labels in a list.    Figure 4.11 Selecting Rows by Multiple Row Labels |

**Read**

Refer to the links below for more details and examples on the attributes .set\_index() and .reset\_index() of the pandas package:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set\_‌index.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html)

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌reset_index.html>

**Read**

Refer to the links below for more details and examples on the methods .loc() and .iloc() of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.‌html>

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.‌html>

### 2.3 Selecting Cells by Positions and Indices

To select elements in the DataFrame, we can specify both column and row labels in the .loc attribute, or the positions in the .iloc attribute, or a combination of both.

Below is a syntax that uses only the row and column indices for the cell selection.

|  |
| --- |
| DataFrame\_name.iloc[row\_start:row\_end, col\_start:col\_end] |

We can also select cells by referring to the corresponding row and column labels.

|  |
| --- |
| DataFrame\_name.loc[["row\_labels"], ["col\_labels"]] |

If we want to select the rows by index but the columns by labels, we can use the index operator and .iloc attribute together.

|  |
| --- |
| DataFrame\_name[["col\_labels"]].iloc[row\_start:row\_end] |

But if we want to select the columns by index but the rows by labels, we need to use both the .loc and .iloc attributes.

|  |
| --- |
| DataFrame\_name.loc[["row\_labels"]].iloc[:, col\_start:col\_end] |

While putting the row labels in the .loc attribute, we need to be aware that the .iloc attribute requires both the row and column indices. Since we do not intend to select the rows by index, we can use the open-end index 0: or simply : here.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to select the first five observations of the first five variables.    Figure 4.12 Selecting Cells Using Indices  Such a selection only makes sense when we exactly know the positions of the variables and observations. More common is the selection of cells based on labels. Suppose we would like to select the observed values in workclass and income for all observations younger than 30 and older than 60 years old.    Figure 4.13 Selecting Cells Using Row and Column Labels  Nevertheless, this selection method only works if the row labels are set and we can refer to them in the .loc attribute. While column labels usually correspond to the variable names, row labels are not necessarily used as row index in most of the datasets. The row positions are therefore more useful in selecting cells in a DataFrame. Suppose we would now like to select the observed values of workclass and income from the first 5 rows.    Figure 4.14 Selecting Cells by Row Indices and Column Labels  If we want to select the observed values of the first two columns from the observations younger than 30 years old, we will have to use the .iloc and .loc attributes at the same time.    Figure 4.15 Selecting Cells by Row Labels and Column Indices |

### 2.4 Selecting Cells by Boolean Masking

In Chapter 2.2 of Study Unit 3, we learned how to use Boolean mask to subset a NumPy array. Here, we will apply the same technique to select cells from a DataFrame. A Boolean mask is an array where each of the values is either True or False. The Boolean mask array is overlaid on top of the data structure that we're querying. And any element aligned with a True value will be selected, and any element aligned with a False value will not.

|  |
| --- |
| DataFrame\_name[Condition] |

We can also create more complex queries by using bitwise logical operators to chain several conditions together.

|  |
| --- |
| DataFrame\_name[(Condition1) &/| (Condition2) &/| …] |

The bitwise logical operators are similar to the logical operators. Instead of writing and/or, we use & (bitwise and), | (bitwise or), or ~ (bitwise not) to combine our conditions in the DataFrame queries. We can also add the bitwise not operator to the above syntax if we want to negate any condition.

We need the bitwise logical operators here because we are actually creating a Boolean mask for each condition within the index operator []. If there are two conditions, two Boolean masks will be compared elementwise by the bitwise operator. The result of this comparison is in turn a Boolean mask as well.

Remember that each Boolean mask/condition needs to be encased in parentheses because of the order of operations.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to select those observations that work more than 40 hours per week.    Figure 4.16 Selecting Cells by a Boolean Mask  The above syntax is a combination of two instructions. First, we create a Boolean mask to select only those observations where census["hours-per-week"] > 40 is True. Recall that census["hours-per-week"] is actually a syntax to select a specific column from the DataFrame. So, the condition here is to tell Python to select the column named hours-per-week first and then assign True to those cases where the observed value is larger than 40. Those row indices where the Boolean mask is True will then be selected from the DataFrame census by the index operator [].  In the next query, we would like to select female respondents from the DataFrame that work more than 40 hours per week.    Figure 4.17 Selecting Cells by Chaining Two Boolean Masks  In the last query, we want to select female or non-white respondents who work more than 40 hours per week.    Figure 4.18 Selecting Cells by Chaining Multiple Boolean Masks  In the first parentheses, we create a Boolean mask for observations where gender is equal to "Female". In the second parentheses, the Boolean mask is created for observations where race is *not* "White". These two masks are compared by the | (bitwise or) operator. The resulting Boolean mask will then be compared by a Boolean mask where the values in hours-per-week are larger than 40. |

## Chapter 3 Merge DataFrames

Lesson Recording - Merge DataFrames in pandas

### 3.1 Appending DataFrames by Rows

It often happens that multiple parties are actually collecting data for the same empirical study simultaneously. Eventually, their collected data must be merged together for analyses. Though the data could be collected at different locations or during different periods, they must consist of the same variables since the study is identical. Merging these datasets means to append their rows below each other to become one dataset.

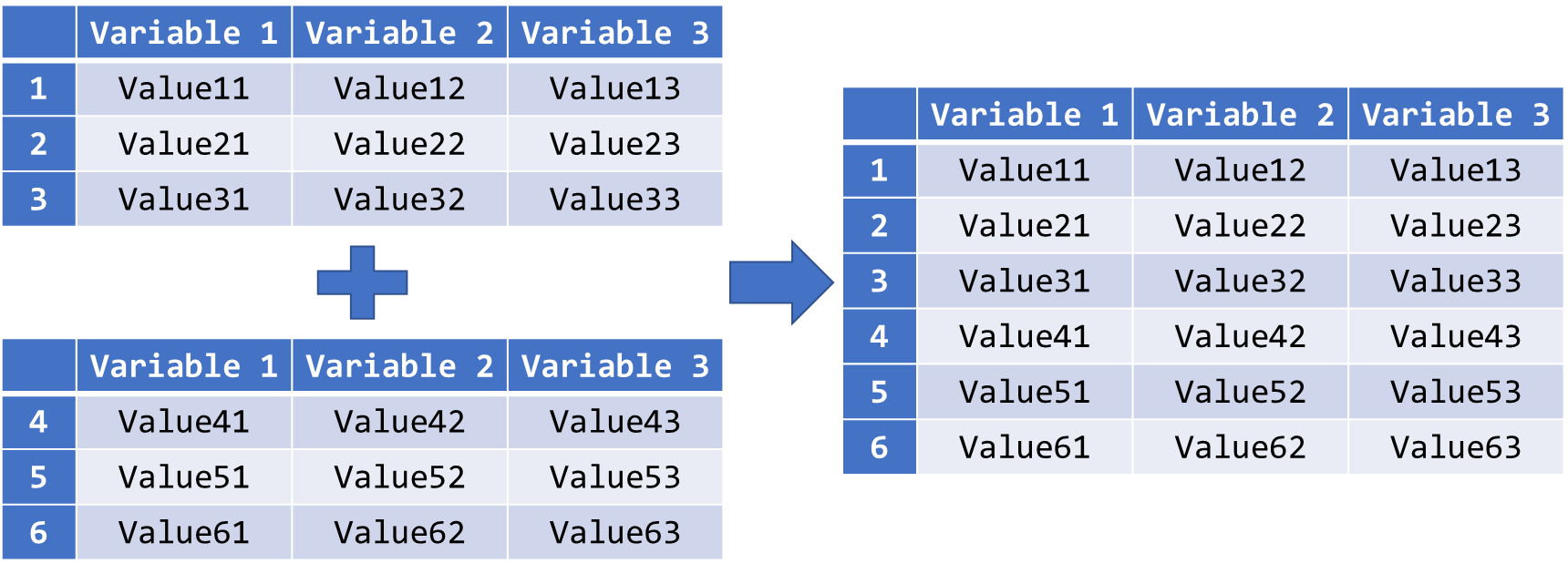


Figure 4.19 Concatenating Two Datasets with Different Rows but Identical Variables

In Python, we can use the .append() method to merge two DataFrames with identical variables into one.

|  |
| --- |
| DataFrame\_name.append(other = [OtherDataFrames]) |

The parameter other is used for the specification of those DataFrames to be appended to DataFrame\_name eventually. If we only have one DataFrame to be assigned to the parameter other, we simply put its name without quotation marks behind other =. In the case of specifying multiple DataFrames to the parameter other, we need to put their names in a list.

|  |
| --- |
| **Example (Cont’d):** In order to study the different income groups in the US census data more efficiently, the data analysts have decided to split the dataset into two. Observatons with income “<50K” will be saved in a new dataset named “census\_low” and those with income “>50K” are now saved in “census\_high”. Now, after the datasets have been cleaned and studied separately, both datasets should be merged again for some joint analyses.    Figure 4.20 Appending Two DataFrames  Note that we can also apply the .append() method on census\_low directly. However, we must be very sure that we no longer need census\_low with its original data since there is no way to retrieve its original content after the appending process, unless we can import the original dataset from an external source again. Furthermore, if we appended census\_high to census\_low directly and re-ran the same code out of whatever reasons, census\_low would eventually contain the observations of census\_high double. Therefore, in Figure 4.20, we first copy census\_low to a new DataFrame named census\_new. And the .append() method is only applied on census\_new. Re-running the same code would not create mess in any of the involved DataFrames at all.  Logically, the observations in the merged DataFrame do not follow the same order as census since it does not play any role in this appending process at all. |

**Read**

Refer to the link below for more details and examples on the .append() method of the pandas package:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame. ‌append.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html)

### 3.2 Merging DataFrames with Different Shapes

The .append() method introduced in Chapter 3.1 is applicable for merging two DataFrames with the same variables by rows. Nevertheless, there are other scenarios when merging multiple datasets in general.

Another rather uncomplicated scenario is that different variables are found across multiple DataFrames. But they contain the same observations.

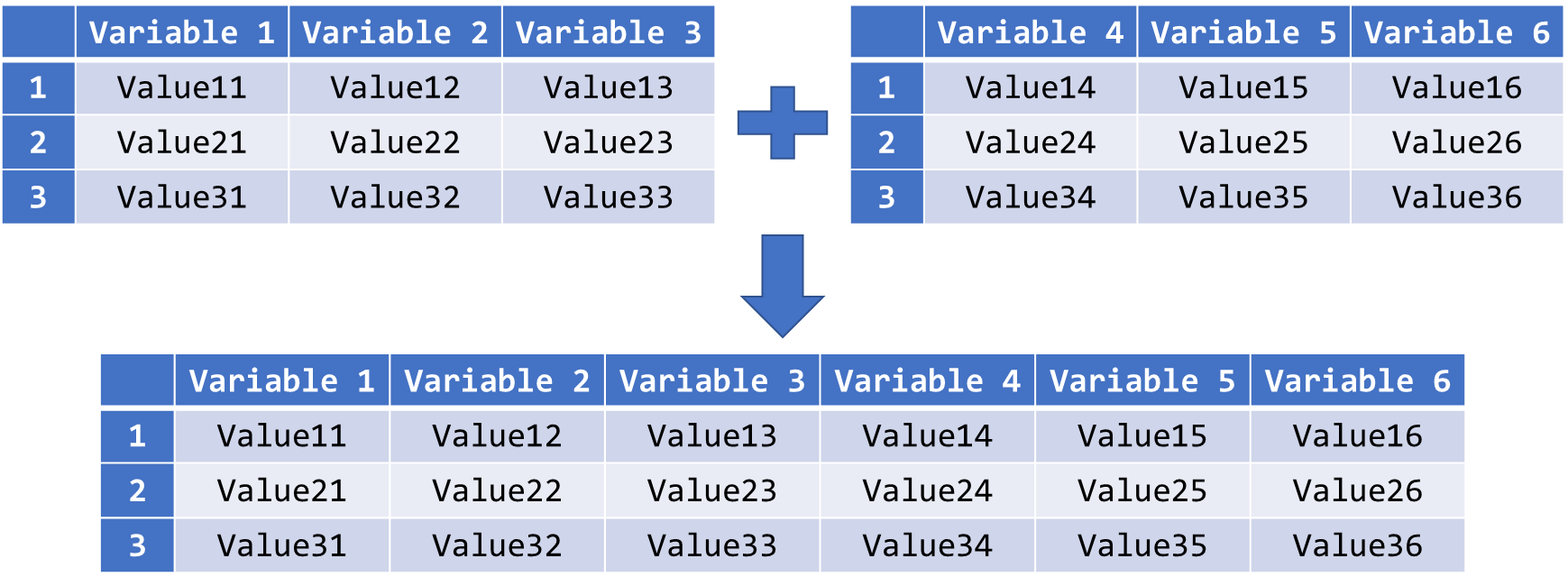


Figure 4.21 Concatenating Two Datasets with Different Columns but Identical Observations

For merging DataFrames by columns, they need to have identical keys, which are usually the row labels of the DataFrames. As Figure 4.21 illustrates, Python can use the row labels of both DataFrames to match identical observations and append their values of all the available variables in both DataFrames in the same row.

A more complicated scenario is that we have multiple DataFrames with some common variables but completely different observations.

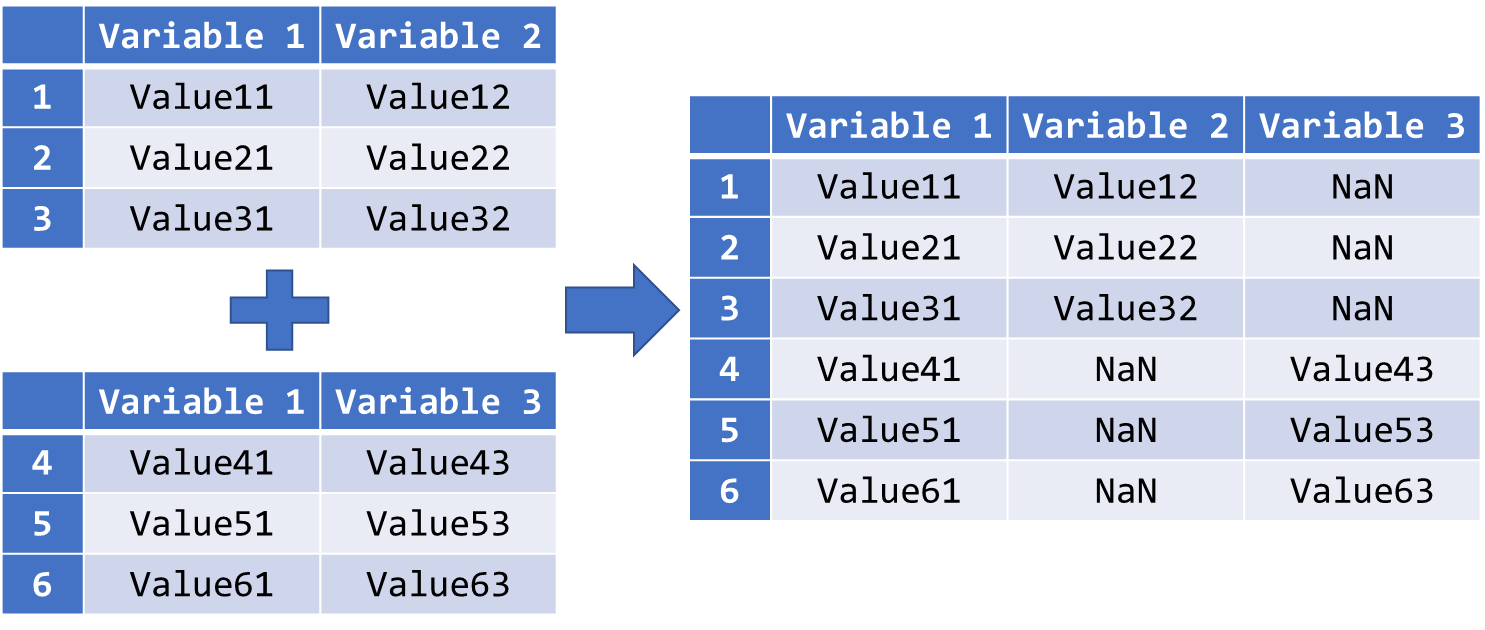


Figure 4.22 Outer Join Two Datasets with Some Common Variables

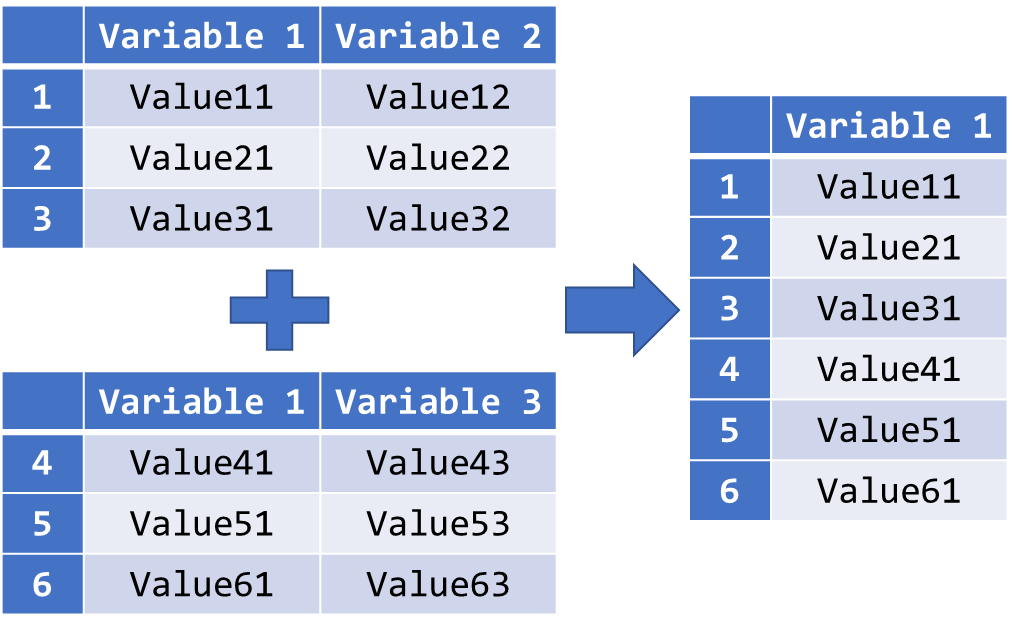


Figure 4.23 Inner Join Two Datasets with Some Common Variables

When merging DataFrames with some common variables, we may obtain two possible results: The output dataset contains either all available columns or only the common variables across all the DataFrames. In Figure 4.22, observation with row label 2 has only got values for Variable 1 and Variable 2. As a result, the value for Variable 3 of this row in the final DataFrame will be a missing value. This type of merging is called the *outer join*. But in Figure 4.23, the final DataFrame only consists of Variable 1 since it is the only common variable in both DataFrames. This type of merging is called the *inner join*.

Similarly, we may also get multiple DataFrames with some common observations but totally different variables.

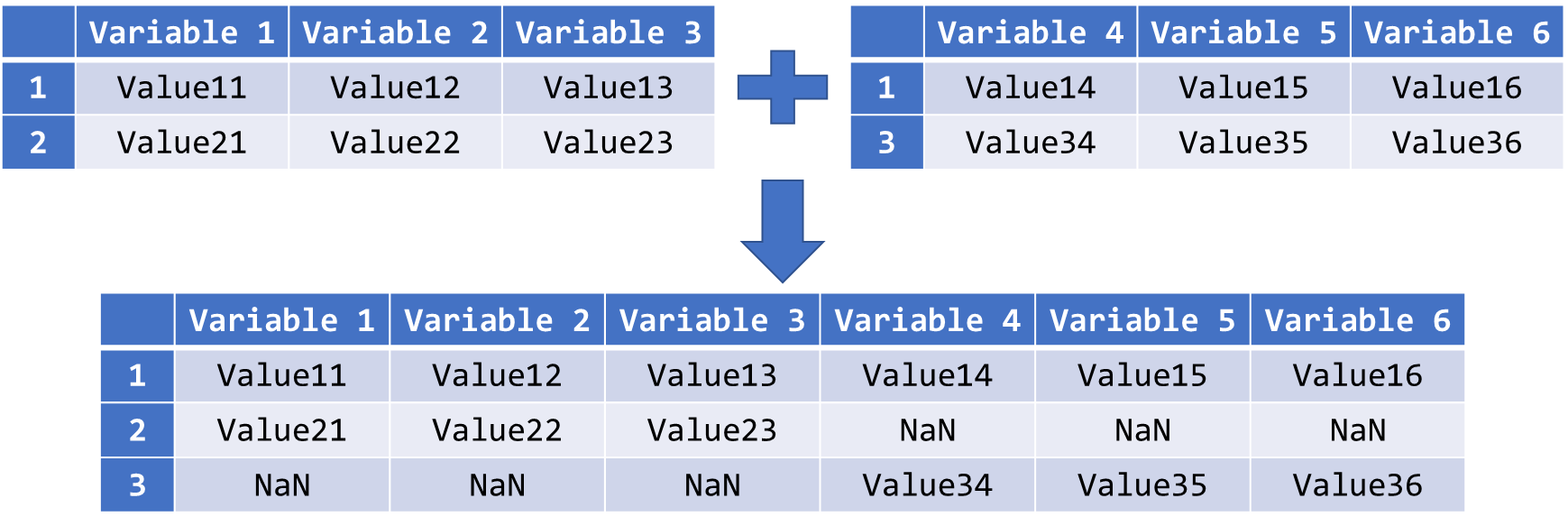


Figure 4.24 Outer Join Two Datasets with Some Common Observations

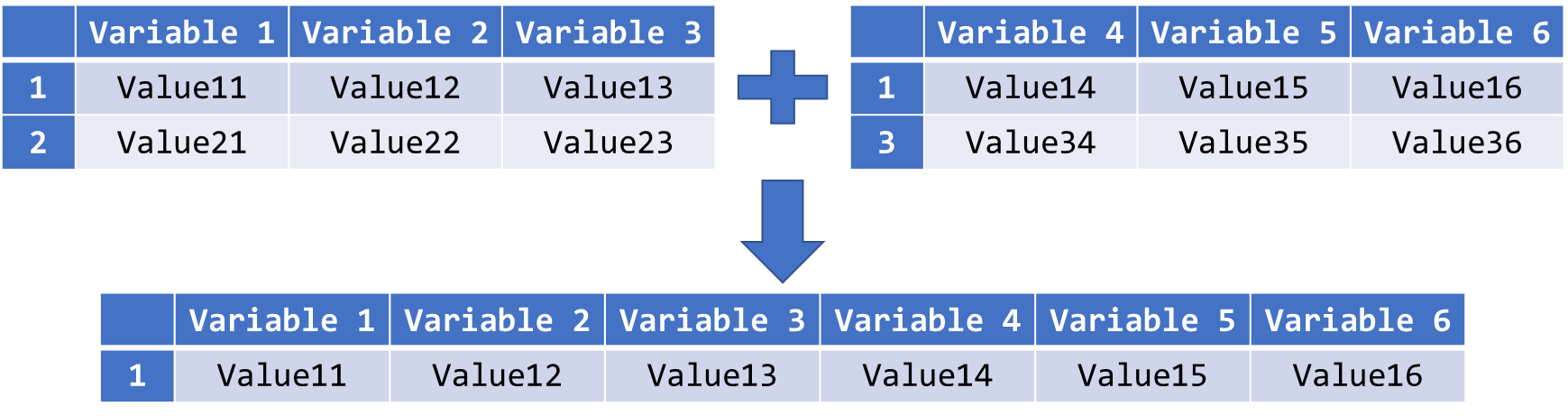


Figure 4.25 Inner Join Two Datasets with Some Common Observations

Same as in Figure 4.22 and Figure 4.23, we have two possible results here too: The output dataset contains either all available rows (outer join) or only the common rows across all the DataFrames (inner join). In Figure 4.24, observation with row label 2 has only got values for Variable 1, Variable 2, and Variable 3. As a result, the values for Variable 4 to Variable 6 of this row in the final DataFrame are entirely missing values. In Figure 4.25, the final DataFrame only consists of observation with ID = 1 since it is the only common observation in both DataFrames.

In the last scenario, the multiple DataFrames to be merged have some common variables and observations. But there are also variables and observations that can only be found in either one of them. If we choose outer join to merge them, the result will be like the output dataset in Figure 4.26.

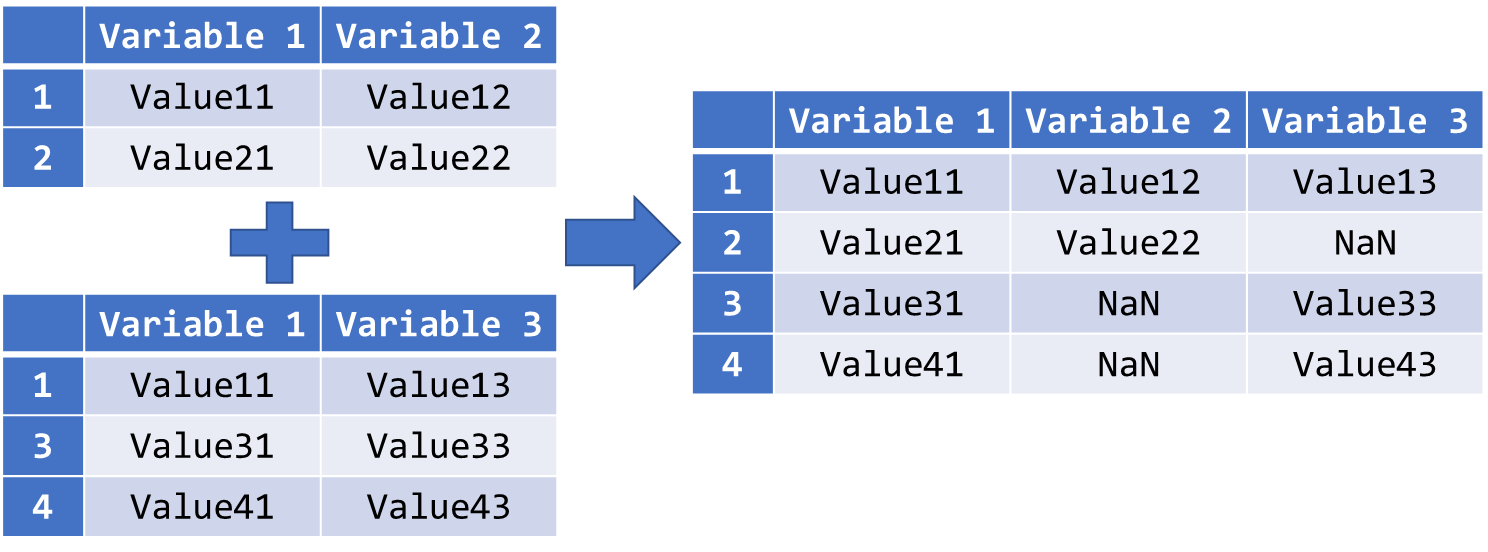


Figure 4.26 Outer Join Two Datasets with Different Shapes

The values of all available cells in either one of the original DataFrames will be taken over in the final DataFrame. Cells that were originally unavailable in both DataFrames such as Value23 will become missing data.

On the other hand, if we choose inner join to merge them, the result will be like the output dataset in Figure 4.27.

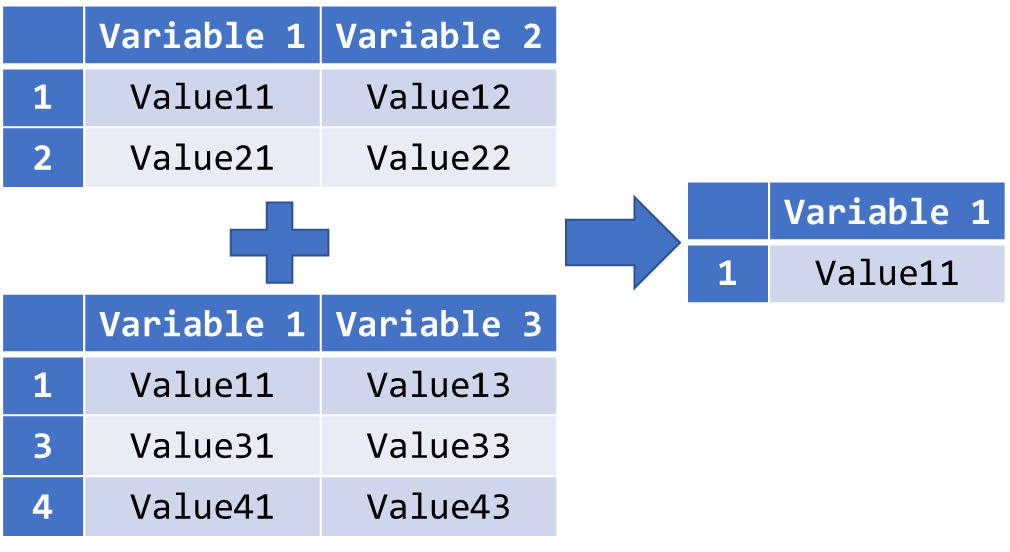


Figure 4.27 Inner Join Two Datasets with Different Shapes

Since Value11 is the only common cell in both DataFrames, it will also be the only cell in the output DataFrame.

In Python, we can use the concat() function to merge multiple DataFrames in all the above-described scenarios. It is a rather complex method, and we will only list out the most commonly used parameters in our syntax introduction. For details, please refer to <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html>.

|  |
| --- |
| finalDF\_name = pd.concat(objs, axis, join) |

The parameter objs is used for the specification of all the DataFrames to be concatenated. Be reminded that we need to put the names of the DataFrames in a list. The parameter axis is the direction along which the concatenation should take place. If axis = 0, the DataFrames will be concatenated below one another, and the concatenation will take place beside one another if axis = 1. The default value here is 0. With the join parameter, we can choose to carry out an outer join or inner join. The possible values here are "outer" and "inner", written as string. If we omit this parameter, “outer” will be considered. The resulting DataFrame will be assigned to the object named finalDF\_name.

|  |
| --- |
| **Example (Cont’d):** We have three sub-DataFrames of the US census study. The first one named census\_ym contains all male observations. However, it has only two variables: gender and income. The second DataFrame named census\_yf has the same variables, but it contains only female observations. The third DataFrame is called census\_x. It contains all observations and all the variables of census except income and gender.  Suppose we concatenate census\_ym and census\_x first. The resulting DataFrame will contain all observations and all variables from the original census dataset.    Figure 4.28 Concatenating Two DataFrames with Different Shapes by Outer Join  Nevertheless, we can see in the fifth row, the value of the observation with row index 4 in the variable gender is NaN. We can conclude that this observation does not exist in census\_ym since it is a female observation.  However, if we concatenate them with inner join, this observation will not exist in the final DataFrame census\_final.    Figure 4.29 Concatenating Two DataFrames with Different Shapes by Inner Join  Figure 4.29 shows that the observation with row index 4 is not included in the final DataFrame census\_final. Furthermore, the number of rows here is 32,650, the number of rows in census\_ym, instead of 48,842, the number of rows in census\_x.  To reconstruct the original census DataFrame, we can first concatenate census\_ym and census\_yf.    Figure 4.30 Concatenating Two DataFrames with Same Variables by Rows  We can see from the output that the resulting DataFrame census\_yg has 48,842 rows, which corresponds to the number of rows in census. The DataFrame census\_yg can now be merged with census\_x.    Figure 4.31 Concatenating Two DataFrames with Same Observations by Columns  As we can see, the rows in the final DataFrame census\_final are sorted by the row indices automatically. |

**Read**

Refer to the three links below for more details and examples on merging DataFrames using the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html>

Refer to the three links below for more details and examples on concatenating DataFrames using the concat() function of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html>

## Chapter 4 Missing Data and Outliers

Lesson Recordings - Missing Data and Outliers in pandas DataFrames

In empirical studies, it often occurs that an observed value of a variable is missing. There are many reasons for missing data: defective measurement tools, withdrawal from the study, refusal of responses to sensitive questions, etc. In Python, there is the NoneType to indicate missing data. Different packages have different ways to display a missing value. For instance, pandas uses a special floating-point value for missing values, and NumPy uses NaN which stands for “Not a Number”.

Missing data are not desirable for data analytics since they cannot be included in constructing models, forecasting, etc. Statistical estimation of parameters can be biased. In pandas, when we use statistical functions on DataFrames, missing values are typically ignored by these functions. As a result, the execution of the code will not be interrupted, but the computation of these functions could be due to unequal underlying sample sizes for each variable.

### 4.1 Identifying Missing Values

Most of the time, we have to work with datasets provided from external sources, and missing values can be referred very differently. The reasons of such discrepancies could be typing errors, or the varying habit of the data collectors when entering missing values, or the limitation of the software used for data entry, etc. In pandas, readers such as the read\_csv() function provide two parameters, na\_filter and na\_values, to convert certain strings to missing values directly while the data are being converted to pandas DataFrame.

|  |
| --- |
| DataFrame\_name = pd.read\_csv("csv\_file\_name.csv", na\_values = "na\_string", na\_filer = True/False) |

The default value of the na\_filter parameter is True. In this case, pandas will convert all white spaces "" to NaN. However, there could be situations where white space is an actual value of interest and not a missing value. The filter should then be turned off and the value would be False.

With the parameter na\_values, we can declare certain strings from our DataFrame to be recognised as missing values. By default, strings like "", "#N/A", "#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN", "-nan", "1.#IND", "1.#QNAN", "N/A", "NA", "NULL", "NaN", "n/a", "nan", "null" are treated as missing values and do not need to be specified explicitly with this parameter.

|  |
| --- |
| **Example (Cont’d):** From Figure 4.1 and Figure 4.2, we can recognise that question marks are used to indicate missing values in the US Adult Census dataset. Suppose we would like to declare every cell that contains a question mark solely as a missing value.    Figure 4.32 Declaring Specific Strings as Missing Values While Importing Data  In the read\_csv() function, we specify a single question mark as string that should be identified as a missing value in the census DataFrame. In Figure 4.32, we can see that the values for workclass and occupation in the fifth row are now NaN, whereas in Figure 4.1, they were simply "?". |

**Read**

Refer to the link below for more details and examples on the parameters associated to missing values in the read\_csv() function of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html>

### 4.2 Locating Missing Values

In Chapter 4.1, we learned that the parameters in the read\_csv() function can instruct Python to indicate missing values clearly with NaN in the DataFrame. Though they will become uniquely identifiable, it is neither easy to locate their positions, nor to detect their existence, if the dataset contains a large number of rows and columns. One way to find out their existence and positions is to count the NaNs in each row and each column.

If the number of NaNs in a column is larger than zero, we have then identified the variables in which missing values exist and including these variables may create biasedness in our analytics tasks. And if the number of NaNs in a variable is large, we can also conclude that the variable may not contain sufficient data for reliable analyses. Equivalently, we can apply the same approach to rows. If the number of NaNs in a row is large, we know that missing values do not only exist for this observation, it may also not be carrying much information for our analyses.

|  |
| --- |
| DataFrame\_name.isnull().sum(axis = 0)  DataFrame\_name.isnull().sum(axis = 1) |

The above syntax is in fact a Boolean masking. It contains two methods of the pandas package. The .isnull() method instructs Python to check every cell of the DataFrame and then return True if it is an NaN. Subsequently, Python should return the sum of each row or each column of the Boolean mask. If the parameter axis is set to 0, the values in a column will be added up together. And if axis = 1, we will obtain the sum of the row instead. The default axis here is 0. Since True is usually represented by 1 and False by 0 when converting a Boolean variable to a numeric value, the sum of a row or a column with only Boolean values will therefore be the same as counting the occurrence of True in it.

If our intension is just to check the existence of missing values, we can use the .any() method instead. The .any() method will return True if at least one of the elements in the array returned by the .isnull() method is True.

|  |
| --- |
| DataFrame\_name.isnull().any(axis = 0)  DataFrame\_name.isnull().any(axis = 1) |

We can retrieve the indices of the rows or columns with missing data by applying the .index method on the resulting object from the syntax above.

|  |
| --- |
| object\_name = DataFrame\_name.isnull().any()  object\_name[object\_name == True].index |

Counting the NaNs in columns has actually a different meaning than counting them in rows. When we count the number of NaNs in columns, we are checking on the existence of missing values in each variable. If they exist, we may need different approaches to adjust the data for different types of variable. For instance, if they exist in a numeric variable, we can replace the missing values by zero or by the mean of the variables. And if a text variable contains missing data, we may add a response category such as “no reply” to it. We can also choose to neglect them if the variable is irrelevant for our analyses of the data.

By counting the NaNs in rows, however, we intend to identify those observations with missing values in at least one of the variables. Depending on the analyses and the importance of the observation, we can choose to delete the observation or to apply the appropriate data adjustments to the affected columns.

|  |
| --- |
| **Example (Cont’d):** From Figure 4.32, we can identify two missing values in workclass and occupation for the fifth observation. Now we would like to find out whether there are more missing values in these two and other variables.    Figure 4.33 Counting the Number of Missing Values in Each Variable  The output shows that workclass, occupation, and native-country are the three variables with missing data. Their proportions of missing data are 2,799 (5.7%), 2,809 (5.7%) and 857 (1.8%) out of 48,842 observations, respectively. We need to further study these observations to conclude on the adjustment we shall apply on the missing values. For this purpose, we shall find out all the observations, or their row indices, with at least one missing value.    Figure 4.34 Identifying Observations with Missing Values  The output generated in Figure 4.34 is rather not satisfactory since we only see the returned Boolean value from the chained methods .isnull().any(axis = 1). Furthermore, it is only useful to us if the rows with missing values are selected from the DataFrame. As a result, we need to filter census with the results above.    Figure 4.35 Selecting Observations with Missing Values from a DataFrame  First, we save the output generated in Figure 4.34 as an object named missrow. After that, we use a Boolean mask to select only those “True” observations from missrow, and the .index method will then return the corresponding row indices to us for selection. Eventually, we apply these row indices to subset census. If we wish to work on this subset of the DataFrame further, we can also assign it to an object in the second line of our code.  Taken from Figure 4.35, there are a total of 3620 rows with at least one missing value in workclass, occupation, and native-country. We can now study the data of these observations and decide on the adjustment measures subsequently. |

**Read**

Refer to the link below for more details and examples on the .isnull() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌isnull.html>

Refer to the link below for more details and examples on the .sum() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sum.‌html>

Refer to the link below for more details and examples on the .any() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.any.‌html>

Refer to the link below for the index() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌index.html>

### 4.3 Replacing Missing Values

After checking the existence of missing values in a DataFrame and locating them, we should decide on how to deal with them. Usually, we can consider deleting the entire observations, replacing them by other values, or simply ignoring them.

To delete an entire row with missing values from the DataFrame, we have two options: the .drop() and .dropna() methods.

|  |
| --- |
| DataFrame\_name.drop(axis = 0, index = [index1, index2, …]) |

With the .drop() method, we can delete an entire row or column by specifying the corresponding indices resulting from the localisation methods introduced in Chapter 4.2. The parameter axis indicates whether rows (0) or columns (1) should be dropped.

The .dropna() method combines the localisation and removal of rows or columns with missing data in a single function. Its usage is rather convenient since we can omit using the .isnull().any() and .index() methods before dropping the corresponding observations or variables.

|  |
| --- |
| DataFrame\_name.dropna(axis = 0, how = "any"/"all") |

The axis parameter in pandas functions or methods should not be a stranger to us anymore. With the how parameter, however, we can instruct Python to drop an observation with only missing values in all variables (all), or to drop an observation with at least one missing value in any variable (any).

The drawback of the .dropna() method is the equal treatment for all missing values throughout the entire dataset. As mentioned in the previous chapters, we have multiple ways to adjust missing data for different types of variable. And depending on the observed values of other variables, we may also want to keep some of the rows with missing data while deleting others.

|  |
| --- |
| **Example (Cont’d):** Now we would like to remove all the rows with missing values using the .drop() method.    Figure 4.36 Dropping Observations with Missing Values Using .drop() Method  The missrow object is still the same one created in Figure 4.35, and the indices assigned to the index parameter in the .drop() method are determined by the same syntax used in the index operator for missrow in the same figure.  We can obtain the same result using the .dropna() function.    Figure 4.37 Dropping Observations with Missing Values Using .dropna() Method |

Another possibility in dealing with missing values is to replace them by a pre-defined value. The most common values used for such purpose are 0 or the variable mean. Some literatures also suggest more sophisticated approaches such as interpolation, extrapolation, or estimation. In Python, the pandas package facilitates replacement of missing values by the .fillna() method.

|  |
| --- |
| DataFrame\_name.fillna(value = repl\_value)  DataFrame\_name["column\_label"].fillna(value = repl\_value) |

Basically, if we apply the .fillna() method on the entire DataFrame, it will replace all missing values that Python could find with the value specified in the parameter. But if we specify a column in the DataFrame and attach the .fillna() method to it, only the missing values found in the corresponding variable will be replaced. By doing this, we can treat missing data in various variable types differently.

|  |
| --- |
| **Example (Cont’d):** Suppose we decide to simply replace all missing values in the DataFrame census by 0.    Figure 4.38 Replacing Missing Values by 0 in the Entire DataFrame  The output shows that all the missing data (NaN) in row 5 are now replaced by 0. Nevertheless, it looks rather odd to have a value 0 in the variables workclass and occupation. As a result, instead of replacing them by 0, we would rather replacing them with the string “Unknown”.    Figure 4.39 Replacing Missing Values by “Unknown” in Specific Columns  In the first line, we apply the same chained methods .isnull().any(axis = 0) on the DataFrame census to detect the existence of missing values in each column. In the second line, we select only those columns where the returned values from the first line are True using the .index method. We then apply the .fillna() method to replace the missing values by “Unknown”, which is the replacement string assigned to the parameter value. |

**Read**

Refer to the link below for more details and examples on the .drop() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.‌html>

Refer to the link below for more details and examples on the .dropna() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌dropna.html>

Refer to the link below for more details and examples on the .fillna() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.‌html>

### 4.4 Detecting and Removing Outliers

Beside missing data, outliers are data that may cause biasedness in the estimation of statistical parameters and hence the goodness of fit of the models. Since biased estimates are undesirable, it is important to identify them and undertake appropriate adjustments before conducting any analysis.

Basically, we can use statistics such as the interquartile range (IQR) to detect the existence of outliers in a variable. Furthermore, visualisation like boxplots or histogram can also be useful to examine the distribution of the variables.

In Chapter 3.2 of Study Unit 3, we have learned how to use the sub-package matplotlib.pyplot to draw histogram in Python. The boxplot() function from the same sub-package facilitates the creation of boxplots for outlier detection.

To compute the interquartile range, we can use the .quantile() method to determine the first and third quartiles of the variable.

|  |
| --- |
| DataFrame\_name["column\_label"].quantile(q = quantile) |

With the parameter q, which is a value between 0 and 1, we can define the quantile of the distribution that the .quantile() method should return to us. Once the 0.25 and 0.75 quantiles of the target variable is obtained, the interquartile range iqr can be computed by iqr = q3 – q1. An observation y is considered as outlier if y < q1 – 1.5 \* iqr or y > q3 + 1.5 \* iqr.

The usual practice in dealing with outliers is to remove them from the dataset. In Python, it suffices to keep observations that do not contain outliers in the target variable. The syntax below generates a subset of rows that do not fulfil the above outlier condition.

|  |
| --- |
| DF[~((DF["Col"] < q1 – 1.5 \* iqr) | (DF["Col"] > q3 + 1.5 \* iqr))] |

Note that DF represents the DataFrame\_Name and Col is the column\_label. The condition left from the bitwise or operator “|” selects all observations with values in “Col“ smaller than q1 – 1.5 \* iqr whereas the condition right from it selects those observations larger than q3 + 1.5 \* iqr. Nevertheless, this would be the combined condition to select all the outliers. To invert the selection, we need to put the bitwise not operator “~” before the entire condition, which must then be put in a pair of parentheses.

|  |
| --- |
| **Example (Cont’d):** In the following, we will use the aforementioned interquartile range rule to detect outliers in the variable hours-per-week, i.e. observations with extraordinary high or low number of working hours.    Figure 4.40 Computing Criteria for Outlier Detection in a Numeric Variable  Based on the results in Figure 4.40, half of the sample works between 40 and 45 hours weekly on average. The corresponding upper and lower thresholds to differentiate outliers from “normal” data are 32.5 and 52.5, respectively.  In the next step, we can select those outlier observations for checking before dropping them from the DataFrame eventually.    Figure 4.41 Selected Outlier Observations from a DataFrame  Before dropping those outlier observations from the DataFrame, we shall actually study them more carefully. For instance, the observation with row index 48829 works for 60 hours per week, which is much higher than the third quartile of the data. But according to the variable workclass, he is self-employed. From this perspective, his average weekly working hours seem sensible. Hence, this observation could be useful for further analyses.  Nevertheless, in order to show how the syntax works, we will still drop all the outlier observations from census that fulfil the above criteria by the following program.    Figure 4.42 Selected Non-Outlier Observations from a DataFrame  The construction of the syntax for selecting those non-outliers is rather straightforward. All we need to do is to place a bitwise not operator “~” in front of the entire selection condition for the outliers that is now wrapped up in a round bracket. |

**Read**

Refer to the link below for more details and examples on the .quantile() method of the pandas package:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame. ‌quantile.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.quantile.html)

## Chapter 5 Data Modification

Lesson Recording - Data Modification in pandas

### 5.1 Sorting Data

The order of the observations in a DataFrame is usually rather arbitrary and random. It can be a result of the sequence in which the data were collected or recorded, or in which they were merged. Sometimes, we may want to sort the data according to values of some variables for better understanding. For instance, we may want to sort an employee dataset by the rank of the employees in the organisation. In Python, the .sort\_values() method from the pandas package helps us to rearrange the order of the rows in a DataFrame.

|  |
| --- |
| DataFrame\_name.sort\_values(by = [List\_of\_var\_names], ascending) |

We can provide a list of variable names to the parameter based on which the DataFrame will be sorted. The sorting hierarchy among these variables drops with the increasing index in the list. If we set the parameter ascending to True, the values of the variables given in the parameter will be sorted in the ascending order, and they will be sorted in the descending order if it is False.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to study the relationship between the individuals’ income and their educational level, which is represented by the numeric variable educational-num in the DataFrame census, as well as their age. The value in educational-num increases with the educational level of the individual. For this purpose, we will sort the DataFrame census first by educational-num in the descending order and then by age in the ascending order. That is, we will see observations with the highest educational level first, and observations with the lowest educational level will appear at the end. And in each educational level, we will first find the youngest individual, and the oldest individual will be put as last in the group.    Figure 4.43 Sorting a DataFrames by Two Variables  In the above syntax, we used two variables for the sorting process. As mentioned, the sorting hierarchy decreases with the index of the variable name in the list. That is, the DataFrame will be first sorted by education-num, followed by age. As a result, we also need a list of two Boolean values to instruct Python on how each of the variables should be sorted. Here, we ask Python to sort education-num in the descending order (False) and then age in the ascending order (True). |

**Read**

Refer to the link below for more details and examples on the sort\_values() function of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_‌values.html>

### 5.2 Discretisation

Sometimes, we need to bin continuous variables into discrete intervals. Through discretisation, the variable could be easier to understand or becomes compatible to some specific analytics models such as decision trees. In the past, we may need to write lengthy programs with various number of if-conditions for this purpose. In Python, we can use the cut() function from the pandas package to discretise continuous variables.

|  |
| --- |
| DataFrame\_name["column"] = pd.cut(x = array, bins, right, labels, include\_lowest, ordered) |

Note that cut() is a function and not a method to be applied on the DataFrame directly. The object left from the equal sign can be any object including a new or an existing column of a DataFrame.

The data to be discretised should be converted to a one-dimensional NumPy array and assigned to the parameter x. With the parameter bins we can specify the number of equal-width bins for the discretisation of the array. But we can also define the bin edges in a numeric tuple or numeric list instead. The parameter right indicates whether the bins should include the rightmost edge or not. If it is False, the leftmost edge will be included instead. Note that one bin edge must be excluded in the discretisation in order not to have overlapping edges. Since the default value here is True, Python usually includes the highest value in the corresponding bin. Therefore, the left edge of the first bin is not included as well by default. By assigning True to the parameter include\_lowest we can instruct Python to include the left edge of the first bin. We can also name the bins by assigning a list of strings to the parameter labels. And they can be ordered if we assign True, the default value here, to the parameter ordered.

|  |
| --- |
| **Example (Cont’d):** In Figure 4.9, we use the age groups categorised by the variable age as the row index of the DataFrame census. There, we wrote a lengthy program to discretise the age into three bins: “Age <30”, “Age 30-59”, “Age 60+”. Here, we can apply the cut() function for the same task. The resulting array will then be assigned to a new variable named “agegroup” in census.    Figure 4.44 Discretising a Numeric Variable into Bins  In the above syntax, we first convert the variable age from the census DataFrame into a NumPy array and assign it to the parameter x. Then we specify in bins the edges of the three bins in a tuple (0, 30, 60, 100), and the right edges should not be included here as we set right = False. These settings enable us to use 30 instead of 29, 60 instead of 59 as bin edges. Since there is no include\_highest parameter for the cut() function, we must set the rightmost edge of the last group higher than the maximum age in our DataFrame. In the final step, we use the same labels as in Figure 4.9 for our bins and specify them as a list for the parameter labels. |

**Read**

Refer to the link below for more details and examples on the .cut() method of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html>

### 5.3 Grouping Data

In data analytics, we often need to group the data by one or more variables and compute the aggregated statistics of some other variables for each group. To group a DataFrame by some variables in Python, we can use the .groupby() method of the pandas package.

|  |
| --- |
| DataFrame\_name.groupby(by = [List\_of\_Labels]).anymethod() |

With the parameter by we can specify a list of column labels, or variable names, based on which the grouping should be conducted. These variables must be categorical so that the number of groups is finite and limited. Attached to the .groupby() method can be any method that we would like to apply on the grouped data. The list of such functions or methods can be found in Table 3.1 of Study Unit 3 since the NumPy functions or methods are also applicable to pandas DataFrames.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to compute the mean of the number of working hours, capital gain and capital loss as well as the age for each age group created in Figure 4.44.    Figure 4.45 Computing the Mean of All Numeric Columns for Grouped Data  The .mean() method of pandas selects all columns of type integer or float and compute their means in each age group. As a result, we obtain the group means of fnlwgt and educational-num as well. To select only the relevant variables for the group mean calculation, we can subset the census DataFrame first in the above syntax.    Figure 4.46 Computing the Mean of Selected Columns for Grouped Data  As a result, the average age of the youngest group is 23.4, while those between 39 and 59 is 42.2 and those who are 60 and older is 66.5. Individuals between the age of 30 and 59 have to work averagely over 43 hours per week while the average of the youngsters and seniors are 36 and 34 hours a week, respectively. Furthermore, seniors at the age of 60 or above have on average the highest capital gain and loss in comparison to the other two groups. |

**Read**

Refer to the link below for more details and examples on the functions groupby() of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌groupby.html>

### 5.4 Transformation, Standardisation, Normalisation

In data analytics, we may need to transform the values of a variable due to various reasons. For instance, we can use the log-transformation to stabilise the variance of a variable, or we need to standardise or normalise variables for customer segmentation analysis when they are measured at different scales and do not contribute equally to the analysis. In Python, we can use various functions to transform, standardise or normalise variables.

The log-transformation of a numeric variable is rather straightforward. Since the log() function is not available in the pandas package, we need to take it from the NumPy package.

|  |
| --- |
| DataFrame\_name["new\_var"] = np.log(DataFrame\_name["var\_name"]) |

It is often useful not to replace the values in the original variable by transformed values since we may still need the original one for other purposes later. As a result, we shall save the transformed values as a new variable in the same DataFrame.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to transform age by the natural logarithm for further analyses such as a Gamma regression.    Figure 4.47 Log-Transformation of a Numeric Variable |

In Python, the standardisation function can be found in the “scikit-learn” package which we will introduce in Study Unit 5. Here, we use the most traditional way to standardise a variable by finding its mean and standard deviation first, and the transformation will be then conducted by a formula.

|  |
| --- |
| var\_mean = np.mean(DF["var\_name"])  var\_std = np.std(DF["var\_name"])  DF["std\_var"] = (DF["var\_name"] – var\_mean) / var\_std |

We can certainly write all the three lines into a single one without assigning the variable mean and variable standard deviation to different variables first. The advantage of splitting such a long syntax into three short ones is the readability of the code and convenience in debugging.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to standardise hours-per-week for further analyses.    Figure 4.48 Standardisation of a Numeric Variable |

Normalisation is another transformation method to scale down a variable. While there are no theoretical upper and lower bounds for standardised variables, the values of a normalised variable can only be in the interval [0, 1]. Same as the standardisation function, the normalisation function in Python can also be found in the “scikit-learn” package. Here, we use the most traditional way to normalise a variable.

|  |
| --- |
| var\_min = np.min(DF["var\_name"])  var\_max = np.max(DF["var\_name"])  DF["norm\_var"] = (DF["var\_name"] – var\_min) / (var\_max – var\_min) |

Same as the syntax for standardisation, we need to find the minimum and maximum of the target variable first and then transform the variable by a formula.

|  |
| --- |
| **Example (Cont’d):** We will now normalise hours-per-week for further analyses.    Figure 4.49 Normalisation of a Numeric Variable  From Figure 4.49, we can clearly see that the values of the normalised variable hours-per-week\_n are entirely non-negative while the standardised variable hours-per-week\_s contains both positive and negative values. |

## Summary

In this unit, we have seen how Python can be used to manipulate, clean, and query data using the pandas package. Querying the DataFrame structures can be done in different ways, such as using the .iloc or .loc attributes for row-based querying or using the square brackets on the object itself for column-based querying. We also saw that one can query the DataFrame through Boolean masking. Furthermore, we also came across situations where we had to use the .append() method and the concat() function to merge multiple DataFrames with different shapes into one. We then explored how to detect and replace missing values as well as outliers in a DataFrame. We also talked about modifying DataFrames for further analyses such as sorting and grouping data, discretising numeric variables to finite number of categories or bins. While pandas offers specific methods or functions such as .sort\_values(), .groupby(), or cut() for these types of data modification, we need to construct our own syntax by combining various NumPy functions to log-transform, to standardise, or to normalise variables if transformation is required for a variable by the analytics methods.

## References

Delve (1996). *The Adult dataset*. The University of Toronto. <http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html>

Kaggle (n.d.). Adult income dataset. Kaggle Inc. <https://www.kaggle.com/wenruliu/adult-income-dataset>

pandas (n.d.). *IO tools (text, CSV, HDF5, …)*. The pandas development team. <https://pandas.pydata.org/docs/user_guide/io.html#io-read-csv-table>

pandas (n.d.). *Merge, join, concatenate and compare*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html>

pandas (n.d.). *pandas.concat*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.‌html>

pandas (n.d.). *pandas.cut*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html>

pandas (n.d.). *pandas.DataFrame.any*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.any.html>

pandas (n.d.). *pandas.DataFrame.append*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.append.html>

pandas (n.d.). *pandas.DataFrame.drop*. The pandas development team. [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.drop. html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.drop.%20html)

pandas (n.d.). *pandas.DataFrame.dropna*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.dropna.html>

pandas (n.d.). *pandas.DataFrame.fillna*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.fillna.html>

pandas (n.d.). *pandas.DataFrame.groupby*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.groupby.html>

pandas (n.d.). *pandas.DataFrame.iloc*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.iloc.html>

pandas (n.d.). *pandas.DataFrame.index*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.index.html>

pandas (n.d.). *pandas.DataFrame.isnull*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.isnull.html>

pandas (n.d.). *pandas.DataFrame.loc*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.loc.html>

pandas (n.d.). *pandas.DataFrame.quantile*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.quantile.html>

pandas (n.d.). *pandas.DataFrame.reset\_index*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.reset_index.html>

pandas (n.d.). *pandas.DataFrame.set\_index*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.set_index.html>

pandas (n.d.). *pandas.DataFrame.sort\_values*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.sort_values.html>

pandas (n.d.). *pandas.DataFrame.sum*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.sum.html>

pandas (n.d.). *pandas.read\_csv*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_‌csv.html>

## Formative Assessment

1. What is not a function/method to display a DataFrame?
2. print()
3. .head()
4. show()
5. display()
6. Which of the following values can be used with .iloc to select rows from a DataFrame?
   1. ["var\_label1", "var\_label2"]
   2. [-4:-1]
   3. ["0", "1", "2"]
   4. DataFrame\_name["var\_label1" == "good"]
7. What is the resulting DataFrame according to the following program?

df = pd.DataFrame([[1, 2]], columns = list('AB'))

df2 = pd.DataFrame([[3, 4]], columns = list('AB'))

df.append(df2)

* 1. A B

1 2

3 4

* 1. A B

3 4

1 2

* 1. A B

1 3

2 4

* 1. A B

3 1

4 2

1. Which columns does dfnew contain according to the following program?

df = pd.DataFrame([[1, 2]], columns = list('AB'))

df2 = pd.DataFrame([[3, 4]], columns = list('BC'))

dfnew = pd.concat([df, df2], join = 'outer')

* 1. A and B
  2. B and C
  3. Only B
  4. A, B, and C

1. Which argument of the pd.read\_csv() function is used to declare specific strings as missing values?
   1. na\_values
   2. na\_filter
   3. na\_drop
   4. na\_omit
2. Which method should be used to replace missing values by another value?
   1. .drop()
   2. .dropna()
   3. .nareplace()
   4. .fillna()
3. Which variable is the lowest in the sorting hierarchy?

df.sort\_values(by = ['Z', 'Y', 'X', 'W'])

* 1. X
  2. W
  3. Z
  4. Y

1. What function of the pandas package is used to discretise numeric variables?
   1. pd.bins()
   2. pd.categorize()
   3. pd.cut()
   4. pd.split()
2. What is not recommended when applying the .groupby() method on a DataFrame?
   1. The grouping variable should be categorical.
   2. The grouping variable should be discretised before.
   3. The grouping variable should be of type float.
   4. The values in the grouping variable can be identical to the row indices.
3. Which functions of the NumPy package do we need to normalise a variable?
   1. np.min() and np.max()
   2. np.mean() and np.std()
   3. np.quantile() and np.range()
   4. np.cov() and np.corr()

## Suggested Solutions

1. What is not a function/method to display a DataFrame?
2. print()

*Incorrect. We can use the print() function to print out a DataFrame.*

1. .head()

*Incorrect. We can use the .head() method to print out the first five rows of a DataFrame.*

1. **show()**

**Correct. show() is a function of matplotlib which is used to show the created graphs.**

1. display()

*Incorrect. We can use the display() function to print out the first five and the last five rows of a DataFrame.*

1. Which of the following values can be used with .iloc to select rows from a DataFrame?
   1. ["var\_label1", "var\_label2"]

*Incorrect. We can only apply .iloc on row positions.*

* 1. **[-4:-1]**

**Correct. -4:-1 are clearly indices representing the row positions to be selected.**

* 1. ["0", "1", "2"]

*Incorrect. The values in the index operator for .iloc must be numeric and not label strings.*

* 1. DataFrame\_name["var\_label1" == "good"]

*Incorrect. The arguments in the index operator for .iloc must be numeric values and not Boolean expressions.*

1. What is the resulting DataFrame according to the following program?

df = pd.DataFrame([[1, 2]], columns = list('AB'))

df2 = pd.DataFrame([[3, 4]], columns = list('AB'))

df.append(df2)

* 1. **A B**

**1 2**

**3 4**

**Correct. The .append() method merges two DataFrames by row.**

* 1. A B

3 4

1 2

*Incorrect. The .append() method appends df2 to df1 and not vice versa.*

* 1. A B

1 3

2 4

*Incorrect. The .append() method does not merge DataFrames by column. Besides, df and df2 are row DataFrames and not column DataFrames.*

* 1. A B

3 1

4 2

*Incorrect. The .append() method does not merge DataFrames by column. Besides, df and df2 are row DataFrames and not column DataFrames. Furthermore, it should be df2 appended to df1 and not df1 to df2.*

1. Which columns does dfnew contain according to the following program?

df = pd.DataFrame([[1, 2]], columns = list('AB'))

df2 = pd.DataFrame([[3, 4]], columns = list('BC'))

dfnew = pd.concat([df, df2], join = 'outer')

* 1. A and B

*Incorrect. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames* *and not only those from df.*

* 1. B and C

*Incorrect. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames and not only those from df2.*

* 1. Only B

*Incorrect. Since it is an outer join and not inner join, the resulting DataFrame should contain all available columns across the original DataFrames and not only the common ones.*

* 1. **A, B, and C**

**Correct. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames, which are A, B and C.**

1. Which argument of the pd.read\_csv() function is used to declare specific strings as missing values?
   1. **na\_values**

**Correct. We can specify a list of strings representing missing values in the DataFrame with na\_values.**

* 1. na\_filter

*Incorrect. na\_filter is used to convert white spaces to missing values.*

* 1. na\_drop

*Incorrect. There is no na\_drop parameter for the pd.read\_csv() function.*

* 1. na\_omit

*Incorrect. There is no na\_omit parameter for the pd.read\_csv() function.*

1. Which method should be used to replace missing values by another value?
   1. .drop()

*Incorrect. The .drop() method is used to delete an entire row/column from a DataFrame.*

* 1. .dropna()

*Incorrect. The .dropna() method is used to delete an entire row from a DataFrame if it contains any missing value.*

* 1. .nareplace()

*Incorrect. There is no method called .nareplace() in the pandas package.*

* 1. **.fillna()**

**Correct. With the .fillna() method, we can replace missing values in a DataFrame by a user-defined value.**

1. Which variable is the lowest in the sorting hierarchy?

df.sort\_values(by = ['Z', 'Y', 'X', 'W'])

* 1. X

*Incorrect. The sorting hierarchy decreases with the increase of the index in the by list. As a result, variable W should be the lowest in the hierarchy.*

* 1. W

**Correct. The sorting hierarchy decreases with the increase of the index in the by list. As a result, variable W is the lowest in the sorting hierarchy.**

* 1. Z

*Incorrect. The sorting hierarchy decreases with the increase of the index in the by list. As a result, variable W should be the lowest in the hierarchy.*

* 1. Y

*Incorrect. The sorting hierarchy decreases with the increase of the index in the by list. As a result, variable W should be the lowest in the hierarchy.*

1. What function of the pandas package is used to discretise numeric variables?
   1. pd.bins()

*Incorrect. There is no function called bins() in the pandas package.*

* 1. pd.categorize()

*Incorrect. There is no function called categorize() in the pandas package.*

* 1. **pd.cut()**

**Correct. We can use the cut() function to discretise numeric variables into bins.**

* 1. pd.split()

*Incorrect. There is no function called split() in the pandas package.*

1. What is not recommended when applying the .groupby() method on a DataFrame?
   1. The grouping variable should be categorical.

*Incorrect. The number of possible values of a grouping variable should be limited. A categorical variable matches this criterion perfectly.*

* 1. The grouping variable should be discretised before.

*Incorrect. If the grouping variable is the result of a discretised numeric variable, then its number of possible values has been limited, which matches the criterion of a grouping variable perfectly.*

* 1. **The grouping variable should be of type float.**

**Correct. This is not recommendable since the number of possible values in a grouping variable should be limited. However, the number of possible values of a float variable is infinite.**

* 1. The values in the grouping variable can be identical to the row indices.

*Incorrect. In the case that the row indices are values of a categorical variable, this categorical variable can perfectly be used as a grouping variable since its number of possible values is limited.*

1. Which functions of the NumPy package do we need to normalise a variable?
   1. **np.min() and np.max()**

**Correct. We need the minimum and maximum of a variable for the normalisation formula.**

* 1. np.mean() and np.std()

*Incorrect. The mean and standard deviation of a variable are used for the standardisation.*

* 1. np.quantile() and np.range()

*Incorrect. The quantile and range functions are not required for normalisation.*

* 1. np.cov() and np.corr()

*Incorrect. Covariance and correlation are not needed for normalisation.*