

Business 4720

## Introduction to Data Management with Python

Joerg Evermann



Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the [Creative Commons by-attribution non-commercial license \(CC BY-NC 4.0\)](https://creativecommons.org/licenses/by-nc/4.0/)

## Learning Goals

After reading this chapter, you should be able to:

- Create and manipulate basic data structures in Python, including lists, tuples, and dictionaries.
- Create and manipulate arrays using the Numpy package for Python, in particular, be able to use slicing to retrieve portions of an array.
- Create and manipulate series and data frames in the Pandas package for Python.
- Compute summary information and to retrieve portions of a Pandas data frame.
- Use Pandas to retrieve information from multiple data frames, including filtering, grouping, and aggregation of information.

## Sources and Further Reading

The material in this chapter draws on the following sources that are valuable for additional information, extended examples, and further details.

Chitlur, Swaroop (2024) A Byte of Python. [HTML Online](#), [PDF](#)

The Python project itself does not provide a tutorial, but dozens can be found on the internet. This open-source book is available in HTML and as a PDF download under a Creative Commons license. It provides an introduction to Python from the very first steps and the basics to creating your own functions, modules, and data structures.

- [NumPy for absolute beginners](#)
- [Quick Start](#)

The NumPy website provides two very good introductions for those new to Numpy. Both are very readable tutorials with many simple, and some not so simple, examples that show the basic functionality of Numpy and will get the reader up to speed on numerical computation and working with array and matrix data in Python.

- [10 Minutes to Pandas](#)

The Pandas website provides very good "10 minute tutorial" that introduces the main concepts and important functions for data manipulation and data summarization in Pandas. It includes sections ranging from basic data structures to viewing and selection of data, merging data frames, grouping, reshaping, plotting and importing and exporting data from and to a variety of formats, covering the essential functionality for doing business analytics in Python with Pandas.

# 1 Introduction

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability through the use of significant<sup>1</sup> whitespace. This unique approach has contributed to Python becoming one of the most popular programming languages in the world.

Python's standard library of functions is large and comprehensive, covering a range of programming needs including web development, data analysis, artificial intelligence, scientific computing, and more. Its simplicity and versatility allow programmers to express concepts in fewer lines of code compared to languages like C++ or Java. Additionally, Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

One of the biggest advantages of Python is its strong community support and the availability of third-party packages, which extend its capabilities even further. Frameworks like Django for web development, Pandas for data analysis, and TensorFlow for machine learning are just a few examples of Python's extensive ecosystem.

Python's popularity can be attributed to its wide range of applications in various fields, such as web development, data science, artificial intelligence, scientific computing, and scripting. It's often used in academic and research settings due to its ease of learning and its ability to handle complex calculations and data manipulation. Major tech companies and organizations use Python, showcasing its reliability and robustness.

In terms of benefits, Python is known for its efficiency, reliability, and speed of development. It is often used for rapid prototyping and iterative development. Python's syntax is clean and its code is generally more readable and maintainable compared to many other programming languages. This readability makes it easier for developers to work on projects collaboratively.

Overall, Python's combination of versatility, simplicity, and powerful libraries makes it a preferred choice for both beginners and experienced developers across diverse fields. Its continued evolution and adaptation to new technologies and paradigms ensure its relevance in the fast-paced world of software development.

# 2 Python versus R

Python and R are two of the most popular programming languages used in data science, each with its unique strengths and applications. Python, known for its general-purpose nature, offers a more comprehensive approach to business analytics, allowing not just data analysis and visualization, but also the integration of data science processes into web applications, production systems, and more. Its simplicity and readability make it

---

<sup>1</sup>"Significant" in this context does not mean lots, it means that spaces at the beginning of a line, that is, line indentations, have meaning and Python code does not work the same way without those spaces.

a go-to language for a wide range of developers, including those who are not primarily data scientists.

Python's extensive libraries like Pandas for data manipulation, NumPy for numerical computations, Matplotlib and Seaborn for data visualization, and Scikit-learn for machine learning make it a powerful tool for business analytics. Moreover, Python's capabilities in machine learning and deep learning, with libraries like TensorFlow and PyTorch, make it a preferred choice for cutting-edge applications in AI.

On the other hand, R, originally designed for statistical analysis, is highly specialized in statistical modeling and data analysis. It offers a rich ecosystem of packages for statistical procedures, classical statistical tests, time-series analysis, and data visualization. R is particularly favored for its advanced statistical capabilities and its powerful graphics for creating well-detailed and high-quality plots.

The choice between Python and R often comes down to the specific requirements of the project and the background of the business analytics team. Python is generally more versatile and better suited for integrating business analytics into larger production applications. It is also the more popular choice for machine learning projects. R, meanwhile, is excellent for pure statistical analysis and visualizing complex data sets. It's often preferred in academia and research settings where complex statistical methods are more commonly required.

Both languages have strong community support and a wealth of resources, making them continually evolving tools in the field of business analytics. Many business analysts are proficient in both, choosing the one that best fits the task at hand. In collaborative settings, it's not uncommon to see teams utilizing both Python and R, leveraging the strengths of each to achieve more comprehensive and powerful data analysis outcomes.

### 3 Using Python

The Interactive Python Shell, Jupyter Notebooks, and PyCharm IDE represent different environments for Python development, each with distinct features and use cases.

**Interactive Python Shell** The Python shell is the most basic and straightforward environment for Python programming. Users can type Python code and see the results instantly. The simplicity is the primary advantages of the Python shell. The immediate feedback makes it excellent for experimentation, learning Python syntax, and quick tests. There is no need for creating files or setting up a project environment. This feature is especially beneficial for beginners who are just starting to learn Python, as it provides a straightforward way to test out new concepts and functions without the overhead of more complex development environments. Figure 1 shows a screenshot of the Python shell.

While the Python shell supports all the features of the Python language, it lacks advanced features found in full-fledged Integrated Development Environments (IDEs),

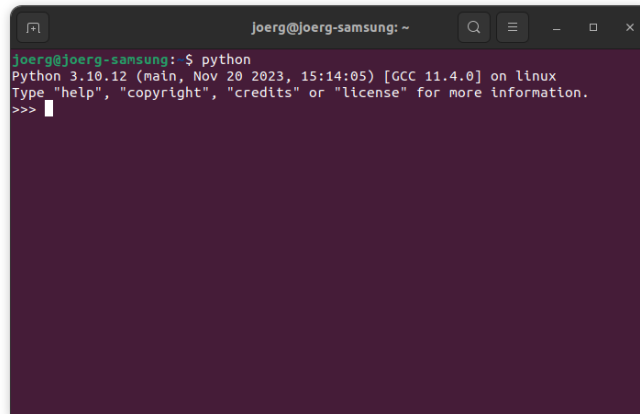
A screenshot of a terminal window titled 'joerg@joerg-samsung: ~'. The terminal shows the command 'python' being executed, which launches the Python 3.10.12 interactive shell. The shell displays the version and build information: 'Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux'. It then prompts the user with 'Type "help", "copyright", "credits" or "license" for more information.' and shows the interactive prompt '>>>' with a cursor.

Figure 1: The Interactive Python Shell

such as code completion, debugging tools, or project management, which are essential for larger projects. Its simplicity is both a strength and a limitation: while it is easy to use, it might not be the best choice for larger programming projects.

On Ubuntu Linux, simply type `python` in a terminal window to launch the Python shell. On Windows and MacOS systems, you will find applications to launch the Python shell in a window. The shell prompts you for commands with the `> > >` prompt. Simply enter the command and press the **ENTER** key to execute a command. Use the `quit()` function to exit the shell. The Python shell remembers your previous commands, so you can use the up and down arrow keys to recall commands and edit them. The Python shell also performs code completion using the **TAB** key, which helps speed up coding and reduce typing errors. Similar to an R session, you should use a notepad editor application to assemble commands and then copy/paste them into the Python shell, as copy/paste results into a notepad editor. This makes editing long commands easier and ensures that your analysis will be repeatable.

### Tips for working efficiently with Python:

To make using Python more efficient, consider doing the following:

- Use the `up-arrow` key to retrieve earlier commands.
- Use the `TAB` key to auto-complete a command.
- Use a notepad app to assemble and edit your commands easily, then copy/paste to Python for execution.
- Use a notepad app to store your results, copy/paste from Python.
- The Ubuntu terminal uses `SHIFT-CTRL-X`, `SHIFT-CTRL-C`, `SHIFT-CTRL-V` for cut/copy/paste.
- Use multiple terminal and Python windows (e.g. one for executing commands, one for reading help documentation or for listing files).
- Don't update packages in the middle of a project.
- Ensure you have a *repeatable, automatable script* for your entire data analysis at the end of a project.

**Jupyter Notebooks** Jupyter notebooks offer a more interactive and versatile platform. Jupyter Notebooks allow users to create and share documents that can contain "cells" where each cell may contain Python code, text (using the Markdown text markup language<sup>2</sup>), equations (using LaTeX), or visualizations. This mix of Python code, documentation, description, and results makes it ideal for data exploration, visualization, and complex analyses where explaining the process is as important as the code itself, allowing for a narrative approach to coding. While Jupyter Notebooks support various programming languages, they are predominantly used with Python. Figure 2 shows a screenshot of a Jupyter notebook in the JupyterLabs Desktop environment.

The immediate feedback upon code execution helps in quick hypothesis testing and data manipulation. Furthermore, the integration of rich media alongside code makes Jupyter Notebooks an excellent tool for creating comprehensive documentation, tutorials, and educational materials.

Notebooks can be easily saved and shared, making them popular in collaborative projects. The ability to see the code, along with its output and accompanying explanation, in a single document enhances understanding and teamwork. Jupyter Notebooks run in a web browser, offering platform independence and eliminating the need for complex software application setups.

When working with Jupyter Notebooks, the term "kernel" denotes a particular version of the Python programming language and environment (i.e. Python packages, etc.) that runs your code. You can enter code in an empty cell and press `CTRL-ENTER` to execute code in the cell. A cell can contain multiple lines of code. Jupyter Notebook cells can be merged, split, moved, copied, and deleted, and you can save, import, and export notebooks, among much other, advanced functionality.

---

<sup>2</sup><https://www.markdownguide.org/>

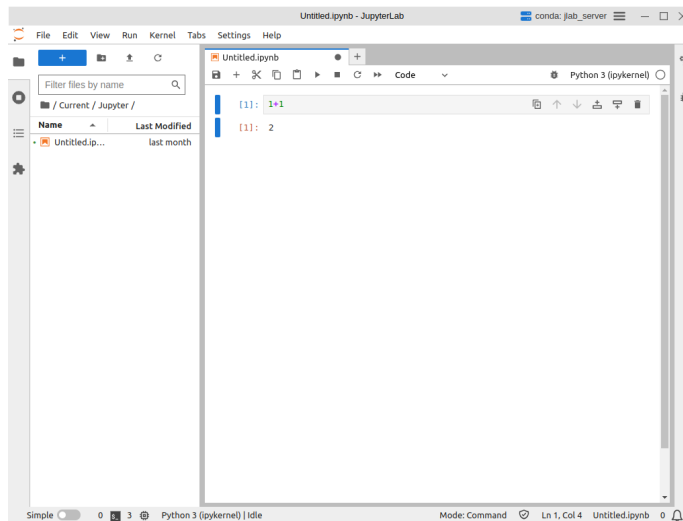


Figure 2: Jupyter Notebook

**PyCharm IDE:** The PyCharm Integrated Development Environment (IDE) is a full-featured software development environment designed specifically for Python. It offers a wide range of tools and features for professional software development, including code completion, debugging, project management, version control integration, and a powerful code editor. PyCharm is more suited for larger and more complex software projects. Its sophisticated environment, while powerful, might be overwhelming for beginners or for those who require a simple platform for exploratory data analysis. Figure 3 shows a screenshot of the PyCharm IDE.

One of the key strengths of PyCharm is its intelligent code editor, offering features like code completion, code inspections, and automated refactoring. These features greatly enhance productivity and reduce the likelihood of programming errors. Additionally, PyCharm includes an integrated debugger and testing support, simplifying the process of diagnosing and fixing issues in programming code. The IDE also offers seamless integration with version control systems like Git, which is essential for collaborative development and code management.

In summary, while the Interactive Python Shell is best for quick, simple tasks and learning the basics, Jupyter Notebooks are ideal for business analytics projects that benefit from an interactive, explanatory, and exploratory approach. PyCharm is the most suitable for comprehensive software development, offering robust tools and features for managing complex codebases. The choice among these depends largely on the specific requirements of the project and the preferences of the developer.

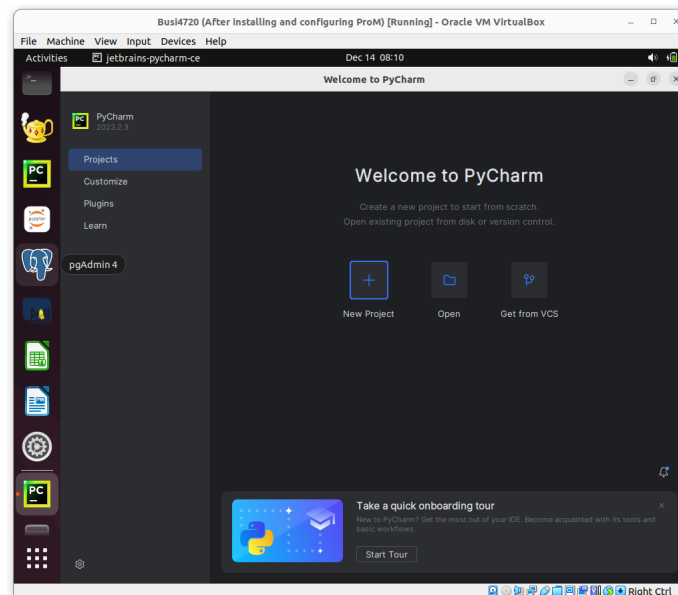


Figure 3: PyCharm Integrated Development Environment (IDE)

## 4 Python Basics

As with R, you can use Python interactively as a calculator. It provides the usual arithmetic, comparison and boolean logic operators. The `//` operator is for integer division with floor (rounding down), the `%` operator is the modulus (remainder) operator. Boolean values are `True` and `False` in Python and *cannot* be abbreviated (unlike in R). The following Python code block illustrates typical usage:

```
# Addition
2 + 2
# Exponentiation
2**4
# Integer division
13 // 3
-13 // 3
# Modulus (remainder)
13 % 3
-25.5 % 2.25
# Comparisons
3 < 5
3 > 5
3 == 5
# Logical and, or, not operators
(3 < 5) and (4 < 2)
(3 < 5) or not (4 < 2)
```



The `print` function in Python is very versatile and provides different ways to print the values of multiple variables. In particular, character strings have a `format` function that can be used to substitute the `{ }` placeholders with values, either by index/number, by name, or by position, as shown in the following Python code block that defines two variables, `age` and `name` and prints their values in a variety of ways:

```
# Define some variables
age = 19
name = 'Malina'

# Print them in different ways.
# Pick your favourite and stick with it.
print('{0} is {1} years old'.format(name, age))
print('{name} is {age} years old'.format(name=name, age=age))
print('{} is {} years old'.format(name, age))
print(f'{name} is {age} years old')
print(name+' is '+str(age)+' years old')
```

Because Python commands can get quite long, Python allows for backslashes to break long lines and continue the command on the following line. While not needed in this case, the following code block illustrates how to use them:

```
print('This is a very long \
string and needs a second line')
i = \
5
print(i)
```

The next code example shows some useful string functions. The `startswith()` function does what its name suggests and returns a boolean (True or False) value. The `find()` function returns either the first position of a string in another string, or -1 if the string is not found.

```
language = 'Innuktitut'
# Check the start of a string
if language.startswith('Innu'):
    print('Yes, the string starts with "Innu"')
# Check if letter contained in string
if 'u' in language:
    print('Yes, it contains the string "u"')
# Find the index of a string in another string
# Returns -1 if not found
if language.find('nuk') != -1:
    print('Yes, it contains the string "nuk"')
```

**Important:** Note the use of leading whitespace or indentation in the lines after the `if` statement in the above code. In Python, this *whitespace is required for defining the program logic!* In the above example, the indented lines indicate the extent of the program block to be executed after the `if` statement. The normal leading whitespace is four spaces.

The `join()` and `split()` functions for character strings do as their names suggest and work with Python lists, illustrated in the code block below:

```
# Join a list of strings with a delimiter
delimiter = '._*'
mylist = ['Nain', 'Hopedale', 'Makkovik', 'Rigolet']
mystring = delimiter.join(mylist)
print(mystring)

# Split a string on a delimiter
thelist = mystring.split(delimiter)
print(thelist)
```

*Lists* in Python are ordered collections of items, and use square brackets `[]` as delimiters. Lists are mutable, i.e. their contents can be changed. Lists may contain items of different data types, including other lists or structured data types. Useful list functions are `len()` which returns the number of items in a list, `append()`, which adds items to the end of the list, and `sort()`, which sorts by value (only for compatible data types in the list). Items can be removed by position using the `del()` or by value using the `remove()` functions.

```
# Define list (Inuit deities)
gods = ['Sedna', 'Nanook', 'Akna', 'Pinga']

# Length of a list
len(gods)
# Iterate over items
for item in gods:
    print(item, end=' ')

# Append to a list
gods.append('Amaguq')
# Sort a list
gods.sort()
# Retrieve items from list
olditem = gods[0]
# Delete item from list
del gods[3]
gods.remove('Sedna')
```

The above example also shows iteration (“repeating”) in Python with the `for` statement. Similar to the earlier example illustrating the `if` statement, note the required

indentation (leading whitespace) in the line(s) after the `for` statement to indicate the extent of the code block that is repeated.

*Tuples* in Python are also ordered collections of items, but they are immutable, i.e. their contents cannot be changed. Tuples use round brackets `()` as delimiters.

```
# Define a tuple (Inuit Nunangat)
regions = ('Inuvialuit', 'Nunavut', 'Nunavik', 'Nunatsiavut')

# Length of a tuple
len(regions)

# Create a tuple of tuples, NOT flattened
more_regions = ('Kalaallit', 'Inupiaq', regions)

# Retrieve element 1 of element 3 in tuple
more_regions[2][1]
```

**Important:** Indexing in Python is zero based, that is, the first element in a list or tuple is number 0, while the last element is number `len() - 1`. This is in contrast to R, where indexing starts at 1.

*Dictionaries* (or short, "dicts") in Python are key-value pairs that map one element to another. In other programming languages, this data structure is also called a *map* or an *associative array* (because it associates keys with their values). Python uses curly brackets `{}` as delimiters; the keys and values are separated using `:`. The value for a key is retrieved using the square bracket operator `[]`. Keys and values may be any data type.

```
# Define a dict (Largest cities of Inuit regions)
c = {
    'Inuvialuit': 'Inuvik',
    'Nunavut': 'Iqaluit',
    'Nunavik': 'Kuujuuaq',
    'Nunatsiavut': 'Nain'
}
```

Keys and values can be retrieved separately using the function `keys()` and `values()`. Dicts are mutable, as the following example shows by removing an entry with `del` and adding another entry.

```

# Get the list of keys
list(c.keys())
# Get the list of values
list(c.values())

# Number of entries in dict
len(c)

# Retrieve a value for a key:
c['Nunavik']

# Delete a key-value pair
del c['Nunavut']

# Add a key-value pair
c['Nunavut'] = 'Iqaluit'

# Check for existence of a key
if 'Nunavut' in c:
    print("\nNunavut's largest city is", c['Nunavut'])

```

A useful function to create dicts from two lists is the `zip()` function, shown below. The `zip()` function creates an iterator over fixed-length tuples that are passed into the dictionary creation function `dict()` as key–value tuples:

```

# Define a list of towns
towns = ['Hopedale', 'Makkovik', 'Nain', 'Postville', 'Rigolet']
# Define a list of population numbers
pops = [596, 365, 1204, 188, 327]
# Create a dictionary
pop_by_town = dict(zip(towns, pops))

```

In Python, lists, tuples, and character strings are examples of *sequences*. All sequences provide membership tests using `in` or `not in` operators, as shown in some of the examples above. Sequences also provide integer indexing and slicing. Note that the end index in a slicing expression is *not inclusive*, that is, the slice extends up to but does not include the final index. This makes it easy to write a slice like `[:len(a)]` where `a` is some sequence (rather than having to write `[:len(a)-1]` as one would in R or other programming languages where the end index is inclusive).

The following code shows some examples for slicing tuples. Note the negative end index in the third example. A negative end index iterates from the end of a sequence forwards”. The slice `regions[1:-1]` extends from the second element to the third of the four elements.

```
# Define a tuple
regions = ('Inuvialuit', 'Nunavut', 'Nunavik', 'Nunatsiavut')
# Slicing of a tuple
regions[1:3]
regions[2:]
regions[1:-1]
regions[:]
```

When slicing in Python, the step size can also be specified, as shown in the next Python code block. The final example slices backwards.

```
# Slicing with step size
regions[::1]
regions[::2]
regions[::3]
regions[::-1]
```

In the above example, pay careful attention to the use of negative indices in the slicing expressions, both for the index as well as the step size.

#### Hands-On Exercise

1. Create a *list* containing the numbers 1 to 10. Use list slicing to create a sublist with only the even numbers.
2. Using a `for` loop, sum all the items in the list.
3. Using a `for` loop, iterate over the list and print each number squared.
4. Write a program to append the square of each number in the range [1:5] to a new list.

#### Hands-On Exercise

1. Create a *tuple* with different data types (string, int, float).
2. Demonstrate how tuples are immutable by attempting to change its first element.
3. Write a program to convert the tuple into a list.

<code>ndarray.ndim</code>	Number of axes
<code>ndarray.shape</code>	Type describing the size of each axis (dimension)
<code>ndarray.size</code>	Total number of elements
<code>ndarray.dtype</code>	The datatype of the elements
<code>ndarray.itemsize</code>	Number of bytes for each element

Table 1: Attributes of NumPy ndarray

### Hands-On Exercise

1. Create a *dictionary* with at least three key-value pairs, where the keys are strings and the values are numbers.
2. Write a Python script to add a new key-value pair to the dictionary and then print the updated dictionary.
3. Create a nested dictionary, that is, a dictionary whose values are dictionaries, and demonstrate accessing elements at various levels.

## 5 NumPy

NumPy, short for Numerical Python, is an essential package for the Python programming language, widely used for scientific computing and data analysis. It provides numerical arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The cornerstone of NumPy is its "ndarray" (n-dimensional array) object. These arrays are more efficient than Python's built-in lists, especially for numerical operations, due to their fixed type and contiguous memory allocation.

One of the reasons for NumPy's popularity in the scientific and data science communities is its seamless integration with other Python libraries. Libraries like Pandas for data manipulation and analysis, Matplotlib for data visualization, and SciPy for scientific computing all build upon and work in conjunction with NumPy, creating a robust ecosystem for scientific computing tasks.

NumPy ndarrays have a set of useful properties or attributes, summarized in in Table 1. Note that the terminology is "*axes*", rather than "*dimensions*", as in the previous chapter on R, although the `ndim` property of an ndarray uses the term "dimension" in its name.

The following Python code block illustrates the use of these properties. Note the use of the `arange()` function to create a one-dimensional array of 15 numbers (from 0 to 14), that is then `reshaped` into a 2-dimensional array with 3 rows and 5 columns. Rows are axis 0, and columns are axis 1.

```

# Import the numpy package
import numpy as np

# Create an array
a = np.arange(15).reshape(3, 5)
# Examine its properties
a.shape
a.ndim
a.dtype.name
a.size

```

The following Python code block shows element-wise operations and array operations on a NumPy array. Python determines automatically which functions are array functions (like `sum()`) and which ones are element-wise functions (like `sqrt()`). Note the creation of the array with the `array()` function from a list of two tuples. Note also the use of the `axis` parameter in the `max` function to specify whether to aggregate by column (`axis=0`) or by row (`axis=1`). The `axis` parameter can also be applied to other functions like `sum()` or `std()`.

```

# Create an array from Python lists and tuples
b = np.array([(1.5, 2., 3),
              (4.0, 5., 6)])
print(b)

# Elementwise operations
3 * b
b + 5
np.sqrt(b)

# NumPy array functions
np.sum(b)
np.max(b)
# Axis 0 is by column
np.max(b, axis=0)
# Axis 1 is by row
np.max(b, axis=1)
np.std(b, axis=0)
# Transpose
np.transpose(b)
# Cov default by row
np.cov(b)
np.cov(np.transpose(b))

```

To create pre-initialized arrays, NumPy provides two convenience functions to create arrays filled with 0s or 1s:

```

# Create an array of zeros with shape (3,4)
x = np.zeros((3,4))
print(x)

# Create an array of ones with shape (2,3,4)
y = np.ones((2,3,4))
print(y)

```

In a generalization of the slicing expressions for Python sequences, each axis of a NumPy array can be sliced using the `[ : ]` or `[ :: ]` expressions, as shown in the following example of a two-dimensional array. The slicing expressions for different axes are separated by commas.

```

b = np.array([[ 0,  1,  2,  3],
              [10, 11, 12, 13],
              [20, 21, 22, 23],
              [30, 31, 32, 33],
              [40, 41, 42, 43]])

# One element
b[2, 3]
# Multiple rows, one column
b[0:5, 1]
# All rows, one column
b[:, 1]
# Multiple rows, all columns
b[1:3, :]
# Last row
b[-1]

```

NumPy arrays also provide convenient iteration of their rows and their elements. Note the use of the `flat` operator to “flatten” a multi-dimensional array to a single dimension in the code block below.

```

for row in b:
    print(row)

for element in b.flat:
    print(element)

```

NumPy provides an easy way to reshape arrays to any dimension. However, it is important to be aware of where and how the elements move during such a reshape. The order can be specified using an optional argument to `rehshape`; consult the NumPy documentation for details. The following example also demonstrates the use of the default random number generator<sup>3</sup> (`rng`) in NumPy to create an array of shape `(3, 4)` filled with random numbers between 0 and 1.

<sup>3</sup>A random number generator in computer science is always a pseudo-random number generator that



```
# Create 3x4 array of random numbers
a = np.floor(10 * np.random.random((3, 4)))

a.shape
a.flatten()
a.reshape(6, 2)
a.T
a.T.shape
```

The above example uses the `flatten()` function which returns a one-dimensional array. The `T` property returns the transpose of the array (same as the `np.transpose()` function).

The next example illustrates concatenation or stacking operations to stack two arrays either vertically, that is, by row, or horizontally, that is, by column. The arrays must be of compatible shape for these stacking operations.

```
# Create another 3x4 array of random numbers
b = np.floor(5 * np.random.random((3, 4)))

# Vertical stacking
np.vstack((a, b))

# Horizontal stacking
np.hstack((b, a))
```

Arrays can be indexed also by boolean arrays. For example, in the following Python code block, the expression `a < 5` constructs a boolean array whose entries are `True` when the corresponding element in `a` is less than 5. This boolean array is then used to select or index the array `a`:

```
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

# Are entries less than 5?
a < 5
# Entries that are less than 5
a[a < 5]

# Are entries even?
a%2 == 0
# Entries that are even
a[a%2 == 0]
```

creates a sequence of numbers according to a deterministic formula (because computers are deterministic), starting from an initial "seed" number. The sequence is repeatable when beginning with the same seed. A good pseudo-random number generate will create sequences that are indistinguishable from true random numbers, for example, those created by rolling dice.

### Hands-On Exercises

1. Create a four-dimensional array with random numbers in the shape indicated by the last four digits of your student number (if your student number contains a 0, use a 1 instead)
2. Construct a new array by swapping the first half of rows (axis 0) with the second half of rows (axis 0)
3. Calculate all covariance matrices formed by the last two axes of your array. *Tip:* Iterate over the first two axes/dimensions with a `for` loop
4. Subtract the mean of the array from each element in the array (mean normalization)
5. Select all elements that are greater than the overall mean
6. Sort the selected elements from the previous step in ascending order

## 6 Data management with Pandas

Pandas is a Python package widely used in data science, data analysis, and machine learning. It is known for its powerful data manipulation and analysis capabilities. It provides fast, flexible, and expressive data structures designed to make working with structured (tabular, multidimensional, potentially heterogeneous) and time series data both easy and intuitive.

Pandas is useful for data cleaning, data transformation, and data analysis. It offers functions for reading and writing data in various formats such as CSV, Excel, JSON, and SQL databases. The Pandas package simplifies handling missing data, merging and joining datasets, reshaping, pivoting, slicing, indexing, and subsetting data. Its time series functionality is particularly robust, offering capabilities for date range generation, frequency conversion, moving window statistics, date shifting, and lagging.

The library's design and functionality are heavily influenced by data analysis needs in finance, which is evident in its powerful group-by functionality for aggregating and transforming datasets, as well as its high-performance merging and joining of datasets. As part of the broader Python scientific computing ecosystem, which includes libraries like NumPy, Matplotlib, and Scikit-learn, Pandas plays an important role in data analysis and machine learning workflows.

Central to Pandas are two primary data structures, the Series and the DataFrame. A Series in Pandas is a one-dimensional array-like object that can hold any data type, including integers, floats, strings, and Python objects. A DataFrame in Pandas is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

The following Python code constructs a Pandas Series of random numbers. The axis labels of a Series (and a DataFrame) are called "index" and allow one to name the elements. The following example also shows how a Python dict can be converted into a series with named elements.

```

# Import the Pandas package
import pandas as pd

# Create a series from a NumPy array of random numbers
s = pd.Series(np.random.randn(5))
print(s.index)

# Provide indices (labels) when creating the series
s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])
print(s.index)

# Create a series from a Python dict that provides labels and values
d = {"a": 0.0, "b": 1.0, "c": 2.0}
print(pd.Series(d))

# Create a series from a dict, reorder the entries
# Produces NaN for index d
print(pd.Series(d, index=["b", "c", "d", "a"]))

```

Note that in the last line of the above example, renaming or reordering the elements of the Series `d` introduces a NaN element for the index "d", because the dict from which the series was created contains no value for the key "d".

Pandas series behave largely like NumPy arrays, but to access their elements by numerical index, one has to use the `iloc` operator, as shown in the following Python code block. This allows slicing the same way as for Python sequences or NumPy arrays. The following example also shows that Series can behave like a Python dict, in that values for a named index ("key") can be retrieved. Series also provide membership tests for "keys" using `in`.

```

# Series behave like an ndarray
s.iloc[0]
s.iloc[:3]
s[s > s.median()]
s.iloc[[4, 3, 1]]
np.exp(s)

# Series behave like a dict
s['a']
s['e']
'e' in s
'f' in s

```

Pandas *DataFrames* are two-dimensional objects. Their columns may have different data types. Conceptually, DataFrames can be considered as a dict of Pandas Series, as the following example demonstrates.

```

# Create a dict of two Series
d = {
    "col1": pd.Series([1.0, 2.0, 3.0],
                      index=['a', 'b', 'c']),
    "col2": pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=['a', 'b', 'c', 'd'])
}

# Create a dataframe from dict
df = pd.DataFrame(d)

```

Constructing the DataFrame `df` "lines up" the two Series on their common row labels (indices), and will introduce a "NaN" for index "d" in column "col1", because that Series does not contain a value for "d".

Pandas provides some useful functions for getting basic information about the shape, dimensions, and contents of a data frame, as illustrated below. The `info()` function provides information about the columns and their data types, while `head()` and `tail()` print the first and last few lines of a DataFrame, and `describe()` produces summary statistics.

```

# Dimensions (rows, columns)
df.shape

# Row labels (index)
list(df.index)
# Column labels
list(df.columns)

# Information about columns and data types
df.info()

# First few rows
df.head()
# Last few rows
df.tail()

# Summary of data
df.describe()

```

Pandas DataFrames can be indexed by column, by label, by integer location, or by boolean vectors. [Table 2](#) shows an overview of the different methods and their return values.

The following Python code block shows examples of these selection/indexing methods:

Select column	<code>df['colname']</code>	Series
Select row by label	<code>df.loc['label']</code>	Series
Select row by integer location	<code>df.iloc[loc]</code>	Series
Slice rows	<code>df[:,:]</code>	DataFrame
Select rows by boolean vector	<code>df[bool]</code>	DataFrame

Table 2: Methods for indexing Pandas DataFrames

```
# Select one column
df['col1']

# Select multiple columns (list of columns)
df[['col1', 'col2']]

# Select rows by label, returns Series
df.loc['a']

# Select single row by number
df.iloc[2]

# Select single column by number
df.iloc[:,1]

# Select rows 0 to 3, columns 0 to 1
df.iloc[0:4:2, 0:2]

# Select every other row 0 to 3
df[0:4:2]

# Select rows by boolean array
df[df['col1'] > 2]
```

For convenience, NumPy operations can also be used to operate on Pandas DataFrames, which are automatically converted to NumPy ndarrays before and converted back after such an operation.

```
# Elementwise operators
df * 5 + 2
1/df
df**4

# Transpose
df.T

# Using Numpy functions on Pandas data frames
np.exp(df)
np.sum(df[['col1', 'col2']], axis=1)
```

Pandas provides useful functions for basic descriptive statistics and aggregation on DataFrames. The `mean()` function takes as its optional first argument the axis number (0 for aggregating the columns, 1 for aggregating the rows) and can skip missing values when summing. Multiple aggregates can be formed using the `agg()` function. The Python code block below illustrates the use of these functions.

```
# Descriptive statistics
# By column
df.mean(axis=0)
# By row
df.mean(axis=1, skipna=False)

# Aggregation with 'agg', by column
df.agg(['sum', 'mean', 'std'], axis=0)
```

Pandas DataFrame values can be sorted by columns, and the functions `nlargest()` and `nsmallest()` can be used to select a DataFrame with only the `n` smallest or largest values in a given column.

```
# Sort values by columns
df.sort_values(by=['col1', 'col2'])

# Find 3 rows with smallest or largest values by column
df.nsmallest(3, 'col1')
df.nlargest(3, 'col2')
```

A very useful way to identify or select data in a Pandas DataFrame is the `query()` function, which accepts a simplified boolean condition as argument. This allows one to write much shorter and compact selection logic, as shown in the following example. Note the two different forms of the same logical operator `&` and `and`.

```
df = pd.DataFrame(np.random.rand(10, 3), columns=['a', 'b', 'c'])

# Pure python
df[(df['a'] < df['b']) & (df['b'] < df['c'])]

# Shorter with Query
df.query('(a < b) & (b < c)')
df.query('a < b & b < c')
df.query('a < b and b < c')
df.query('a < b < c')
```

Finally, Pandas provides many functions for reading and writing DataFrames from and to a variety of serialization formats and even SQL RDBMS. See the [Pandas IO user guide](#) for details.

## 7 Basic Pandas, Compared to R and SQL

This subsection introduces basic ideas of data manipulation with the Pandas. A simple data set is used to illustrate basic functionality. The example dataset for this section is the Fuel Consumption Ratings for battery electric vehicles, provided the Government of Canada through its Open Government Portal<sup>4</sup>. The dataset contains the variables shown in Table 3.

Column	Data Type	Definition
Make	Categorical (string)	Manufacturer
Model	Categorical (string)	Model name
Year	Numeric	Model year
Category	Categorical (string)	Small, Midsize, Large, Pickup, SUV, Station Wagon, etc.
City	Numeric	Consumption in l/100km equiv.
Hwy	Numeric	Consumption in l/100km equiv.
Comb	Numeric	Consumption in l/100km equiv.
Range	Numeric	Driving range in km

Table 3: Fuel efficiency data set variables

**Reading Data:** The following Python code reads the CSV file into a Pandas DataFrame and examines the data.

```
# Import pandas
import pandas as pd
# Read CSV into a Pandas data frame
data = pd.read_csv('https://evermann.ca/busi4720/fuel.csv')

data.shape
list(data.columns)
data.info()
data.describe()
```

**Filtering:** The easiest way to apply filters to the data is to use the `query()` function in Pandas. This is equivalent to the R `filter()` function. The code block below illustrates this and shows the equivalent R and SQL code:

```
# Filter values
data.query('Make=="Ford" & Year==2023')
```

<sup>4</sup><https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64>

Equivalent in R:

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  print()
```

Equivalent in SQL:

```
SELECT *
FROM data
WHERE Make=='Ford' AND
      Year==2023;
```

**Selection:** One or more columns can be selected from the data frame using the data frame indexing described above. This is equivalent to the `select()` function in R:

```
# Filter values and select columns
data.query('Make=="Ford" & Year==2023') \
  [['Model', 'Category', 'Range']]
```

Equivalent in R

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  select(Model, Category,
         Range) |>
  print()
```

Equivalent in SQL:

```
SELECT Model, Category, Range
FROM data
WHERE Make=='Ford' AND
      Year==2023;
```

**Column Creation:** A new column can be created either by simply declaring/defining it, or, alternatively, with the Pandas `assign()` function, which has the advantage that it returns the updated data frame and can therefore be part of a chain of functions operating on the data frame, as shown in the following example code block. The `assign()` function serves the same purpose as the `mutate()` function in R.

```
# Filter values, create a new calculated column,
# and select some columns
data.query('Make=="Ford" & Year==2023') \
  .assign(HwyRange = data['Range']*data['Comb'] / data['Hwy']) \
  [['Model', 'Category', 'Range', 'HwyRange']]
```

Equivalent in R:

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  mutate(HwyRange=Range*Comb/Hwy) |>
  select(Model, Category,
         Range, HwyRange) |>
  print()
```

Equivalent in SQL:

```
SELECT Model, Category, Range,
       (Range*Comb)/Hwy AS HwyRange
FROM data
WHERE Make=='Ford' AND
      Year==2023;
```



**Renaming:** Renaming of columns is done using the Pandas `rename()` function, which accepts a dict of the old and new column names, as key and value, respectively. For example, the following code renames the Range column as the ComRange column. Because `rename()` returns the data frame, the function can be inserted into the function chain in the following code block:

```
# Filter values, create two new calculated columns,
# rename a column, and select columns
data.query('Make=="Ford" & Year==2023') \
    .assign(HwyRange = data['Range']*data['Comb'] / data['Hwy']) \
    .assign(CityRange = data['Range']*data['Comb'] / data['City']) \
    .rename(columns={'Range': 'ComRange'}) \
    [['Model', 'Category', 'ComRange', 'CityRange', 'HwyRange']]
```

Equivalent in R:

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  mutate(HwyRange =
    Range * Comb / Hwy) |>
  mutate(CityRange =
    Range * Comb / City) |>
  rename(ComRange = Range) |>
  select(Model, Category,
         ComRange, CityRange,
         HwyRange) |>
  print()
```

Equivalent in SQL:

```
SELECT Model, Category,
       Range AS ComRange,
       (Range * Comb) / Hwy
       AS HwyRange,
       (Range * Comb) / City
       AS CityRange
FROM data
WHERE Make=='Ford' AND
       Year==2023;
```

**Distinct Values:** The Pandas function `unique()` and `drop_duplicates()` can be used to identify duplicates. The `unique()` function returns a list of unique values for a column or columns, whereas the `drop_duplicates()` function returns a data frame with duplicate rows dropped. The example below illustrates how this function can be used:

```
# Find distinct values
data[['Make', 'Model']].drop_duplicates()
```

Equivalent in R:

```
data |>
  distinct(Make, Model) |>
  print()
```

Equivalent in SQL:

```
SELECT DISTINCT Make, Model
FROM data;
```

**Ordering:** Ordering or sorting data by values is done using the Pandas function `sort_values()`. This is similar to the R function `arrange()` and allows sorting in ascending or descending order, as shown in the following Python code block:

```
# Filter values, order by values of two columns
# and select columns
data.query('Make=="Ford" & Year==2023') \
    .sort_values(['Category', 'Range'], ascending=[True, False]) \
    [['Model', 'Category', 'Range']]
```

Equivalent in R:

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  select(Model, Category,
         Range) |>
  arrange(Category,
          desc(Range)) |>
  print()
```

Equivalent in SQL:

```
SELECT Model, Category, Range
FROM data
WHERE Make='Ford' AND
      Year==2023
ORDER BY Category ASC,
         Range DESC;
```

**Relocating Columns:** There is no function for relocating columns in the sense of changing the order of columns retrieved from a Pandas data frame. Instead, this is done when selecting the columns to retrieve. Hence, there is no equivalent to the R function `relocate()`.

```
# Filter values, order by values of two columns
# and select columns in particular order
data.query('Make=="Ford" & Year==2023') \
    .sort_values(['Category', 'Range'], ascending=[True, False]) \
    [['Category', 'Range', 'Model']]
```

Equivalent in R:

```
data |>
  filter(Make=='Ford',
         Year==2023) |>
  select(Model, Category,
         Range) |>
  arrange(Category,
          desc(Range)) |>
  relocate(Category, Range) |>
  print()
```

Equivalent in SQL:

```
SELECT Category, Range, Model
FROM data
WHERE Make='Ford' AND
      Year==2023
ORDER BY Category ASC,
         Range DESC;
```

**Grouping and Summarizing:** Similar to the `group_by()` and `summarize()` functions in R to compute aggregate values by group, Pandas provides the analogous functions `groupby()` and `agg()` for this purpose. These are illustrated in the following example:

```
# Filter values, group the data,
# calculate aggregates of multiple columns
# filter on aggregate data, order by value
# and select certain columns
data.query('Year==2023') \
    .groupby(['Make', 'Category']) \
    .agg(meanCity = ('City', 'mean'),
         meanHwy = ('Hwy', 'mean'),
         meanComb = ('Comb', 'mean'),
         maxRange = ('Range', 'max'),
         nVehicle = ('Model', 'count')) \
    .query('nVehicle > 1') \
    .sort_values(['Category', 'meanComb']) \
    .reset_index() \
    [[ 'Category', 'meanComb', 'Make', 'meanCity', \
       'meanHwy', 'maxRange', 'nVehicle' ]]
```

Equivalent in R:

```
data |>
  filter(Year==2023) |>
  group_by(Make, Category) |>
  summarize(
    meanCity = mean(City),
    meanHwy = mean(Hwy),
    meanComb = mean(Comb),
    maxRange = max(Range),
    nVehicle = n()) |>
  filter(nVehicle > 1) |>
  arrange(Category, meanComb) |>
  relocate(Category, meanComb) |>
  print()
```

Equivalent in SQL:

```
SELECT Category,
        AVG(Comb) AS meanComb,
        Make,
        AVG(City) AS meanCity,
        AVG(Hwy) AS meanHwy,
        MAX(Range) AS maxRange,
        COUNT(*) AS nVehicle
FROM data
WHERE Year==2023
GROUP BY Make, Category
HAVING COUNT(*) > 1
ORDER BY Category ASC,
         meanComb ASC;
```

## 8 Advanced Pandas for Data Analysis

This section the use of Pandas for descriptive data analysis using the Pagila database data as an example. The Pagila database<sup>5</sup> is a demonstration database originally developed for teaching and development of the MySQL RDBMS under the name Sakila<sup>6</sup>. Pagila is designed as a sample database to illustrate database concepts and is based

<sup>5</sup><https://github.com/devrimgunduz/pagila>,  
<https://github.com/devrimgunduz/pagila/blob/master/LICENSE.txt>

<sup>6</sup><https://dev.mysql.com/doc/sakila/en/>,  
<https://dev.mysql.com/doc/sakila/en/sakila-license.html>

on a fictional DVD rental store. It originally consists of several tables organized into categories like film and actor information, customer data, store inventory, and rental transactions. For this chapter, the Pagila data was summarized in a few related CSV files.

The following Python code block reads the rentals data of the Pagila database into a Pandas DataFrame using the `read_csv()` function. It then converts the data type of some columns from character strings to datetime types so that one can use date and time operations and arithmetic later.

```
rentals = pd.read_csv(
    'http://evermann.ca/busi4720/rentals.csv')
actors = pd.read_csv(
    'https://evermann.ca/busi4720/actors.categories.csv')
addresses = pd.read_csv(
    'https://evermann.ca/busi4720/addresses.csv')
```

When printing DataFrames, Pandas by default abbreviates the output to manageable size. The number of rows and number of columns to be printed is controlled by two Pandas options that can be set as shown in the following example, which removes any limits.

```
pd.set_option('display.max_rows', None)
pd.set_option('display.width', None)
```

The remainder of this section shows how Pandas can be used to provide equivalent results as obtained in the previous chapter using R/dplyr and in the chapter on relational databases with SQL. *Compare the Python code to the R code and the SQL code to achieve similar results.*

**Example:** Find all films and the actors that appeared in them, ordered by film category and year, for those films that are rated PG.

```
data = pd.merge(rentals, actors, on='title',
                suffixes=('_customer', '_actor'), how='outer')
data.query('rating == "PG"') \
    .assign(actor = data['last_name_actor'] + \
                ', ' + data['first_name_actor']) \
    .rename(columns={'release_year': 'year'}) \
    [['actor', 'title', 'category', 'year']] \
    .drop_duplicates(['actor', 'title', 'category', 'year']) \
    .groupby(['category', 'year', 'title']) \
    ['actor'].apply(list) \
    .reset_index() \
    .sort_values(['category', 'year']) \
```

This Python code block above performs a series of data manipulation operations using Pandas. The operations merge, filter, transform, and group data from the Pagila movie rental dataset.

- *Merging DataFrames*: The "rentals" and "actors" data frame are merged based on the "title" column that is common to both. The `suffixes` parameter is used to differentiate columns with the same name in both data frames, by adding either "\_customer" or "\_actor" to those column names. The `how='outer'` parameter ensures that all records from both data frames are included in the result.
- *Filtering Data*: After merging, the code filters the resulting data frame using the `query()` function to include only rows where the "rating" column contains the value "PG".
- *Creating a New Column*: A new column, "actor", is created with the `assign()` function by concatenating the "last\_name\_actor" and "first\_name\_actor" columns, separated by a comma.
- *Renaming a Column*: The "release\_year" column is renamed to "year" using the `rename()` function.
- *Selecting and Rearranging Columns*: The data frame is then reduced and rearranged to include only the columns "actor", "title", "category", and "year".
- *Dropping Duplicates*: Duplicate rows based on the combination of "actor", "title", "category", and "year" are removed. This ensures that each combination is unique in the dataset.
- *Grouping Data and Creating a List*: The data is grouped by "category", "year", and "title" using the `groupby()` function. For each group, the "actor" values are aggregated into a list. This creates a list of actors for each movie title, categorized by year and category.
- *Resetting Index*: After the grouping and aggregation, the index is reset to turn the grouped data back into a regular data frame.
- *Sorting Data*: The data frame is then sorted by "category", then "year", and finally "title" using the `sort_values()` function.

**Example:** Find the most popular actors in the rentals in each city.

The Python code block below combines the data frames from the multiple CSV files that make up the Pagila data set, because the combined, full data is used for other analysis examples below.

- The Python code block below merges the "rentals" data frame with the "addresses" data frame based on the columns "customer\_address" in "rentals" and "address\_id" in "addresses".
- The following command then merges the resulting data frame with the "actors", based on the "title" column.

```

full_data = pd.merge(rentals, addresses,
                     left_on='customer_address',
                     right_on='address_id')
full_data = pd.merge(full_data, actors, on='title',
                     suffixes=('_customer', '_actor'))

```

The following Python code block performs the analysis to find the most popular actors on the full data constructed above, using the following steps:

- Using the `assign()` function, the code creates the "actor" column from last name and first name of the actor, separated by a comma.
- The code then groups the data by "city" and "actor" and calculates the size of each group, that is, it counts how the number of titles in each group (not unique titles). The index is reset to return to an ungrouped data frame.
- The Python code creates a new column "ranking" using the `assign()` function and a lambda function, that is, an inline function without a name. This lambda function groups a data frame by "city", then selects the "count" column for each city and ranks it. The `rank` method is set to 'min', which means actors with the same count will have the same rank, and it ranks in descending order of count.
- The code filters the data frame to select the top 3 actors (or ties) in terms of rental counts in each city using the `query()` function.
- The filtered data is then sorted with the `sort_values()` function by "city", "ranking", and "actor".

```

full_data \
    .assign(actor=full_data['last_name_actor'] + ', ' +
            full_data['first_name_actor'] ) \
    .groupby(['city', 'actor']) \
    .agg(count = ('title', 'count')) \
    .reset_index() \
    .assign(ranking=lambda df:
            df.groupby('city')['count']
              .rank(method='min', ascending=False)) \
    .query('ranking <= 3') \
    .sort_values(by=['city', 'ranking', 'actor'])

```

**Example:** Find the customers who spend the most on rentals, and the number of rentals with the highest total rental payments for each category grouped by rental duration.

```

full_data \
    .assign(customer=full_data['last_name_customer'] + ', ' +
            full_data['first_name_customer'] ) \
    [['customer', 'amount', 'rental_duration', \
      'category', 'phone', 'city']] \
    .groupby(['category', 'rental_duration', 'customer']) \
    .agg(payments=('amount', 'sum'),
        num_rentals=('amount', 'count')) \
    .reset_index() \
    .assign(ranking=lambda df: \
        df.groupby(['category', 'rental_duration'])['payments'] \
        .rank(method='min', ascending=False)) \
    .loc[lambda df: \
        df.groupby(['category', 'rental_duration'])['ranking'] \
        .idxmin() ]

```

By now, it should be clear what most of the functions in the analysis accomplish. The `idxmin()` function within the `loc[]` operator of the dataframe selects the smallest index, i.e. the smallest (highest) ranking when the data is grouped by category and rental duration.

**Example:** Get the top 5 and the bottom 5 grossing customers for each quarter.

This example demonstrates the Pandas date and time function `to_period()`. The argument `Q` returns quarters. Other frequently used arguments are 'D' for days, 'W' for weeks, 'M' for months, 'Y' for years, 'H' for hours, and 'T' for minutes. The use of the `sort_values()` function demonstrates "mixed" sorting, ascending by quarter, and descending by payments.

```

full_data \
    .assign(customer=full_data['last_name_customer'] + ', ' +
            full_data['first_name_customer'],
            q = pd.to_datetime(full_data['rental_date'], utc=True)
                .dt.to_period("Q")) \
    [['customer', 'q', 'amount', 'rental_date']] \
    .groupby(['q', 'customer']) \
    .agg(payments=('amount', 'sum')) \
    .reset_index() \
    .drop_duplicates(['customer', 'q', 'payments']) \
    .assign(rank_top = lambda df :
        df.groupby('q')['payments']
        .rank(method='min', ascending=False),
            rank_bot = lambda df :
        df.groupby('q')['payments']
        .rank(method='min', ascending=True)) \
    .reset_index() \
    .query('rank_top <= 5 or rank_bot <= 5') \
    .sort_values(by=['q', 'payments'], ascending=[True, False])

```

**Example:** Find the set of film titles by rental customer and the total number rentals for each customer.

The code below introduces the `apply()` function. This function applies some other function to every element in a Pandas series or data frame. In the example below, it first used to apply the function `list()` to all elements of the "title" column, creating the "titles" column, that is, a list of titles for each customer. It also applies an unnamed lambda function to all elements of the "titles" column in the grouped data. Here the lambda function converts its input `x` to a set (i.e. it removes duplicates), and then converts the set to a list.

```
full_data \
    .assign(customer=full_data['last_name_customer'] + ', ' +
            full_data['first_name_customer']) \
    [['customer', 'title']] \
    .groupby('customer') \
    ['title'].apply(list) \
    .reset_index(name='titles') \
    .assign(rentals = lambda df :
            df['titles'].apply(len) ,
            unique_titles = lambda df :
            df['titles'].apply(lambda x: list(set(x)))) \
    .drop(columns=['titles']) \
    .sort_values(by='customer')
```

### Hands-On Exercise

1. Find all films with a rating of 'PG'
2. List all customers who live in Canada (with their address)
3. Find the average *actual* rental duration for all films
  - This requires date arithmetic
4. Find the average overdue time for each customer
  - This requires date arithmetic
5. List all films that have never been rented
6. List the names of actors who have played in more than 15 films