Business 4720

Introduction to Data Management with Python

Joerg Evermann

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[1] 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 vast 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.

Intro Tutorial: A very good introduction to Python can be found at https://python.swaroopch.com/, or, as a downloadable PDF, at https://github.com/swaroopch/byte-of-python/releases/

[1] "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.

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 busienss 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 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.

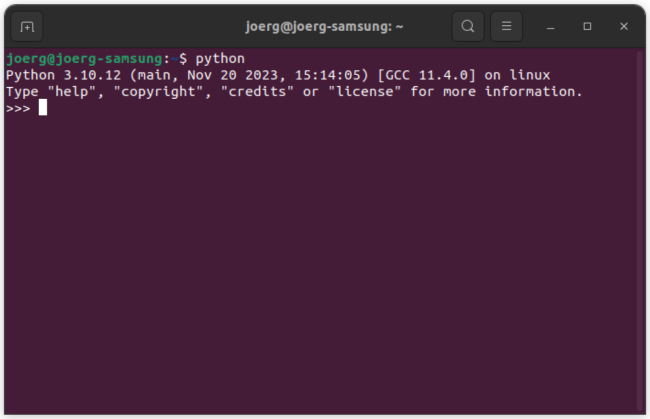
The choice between Python and R often comes down to the specific requirements of the project and the background of the business analytics team. Python is generally more versatile and better suited for integrating business analytics into larger production applications. It is also the more popular choice 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



```
joerg@joerg-samsung:~$ python
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

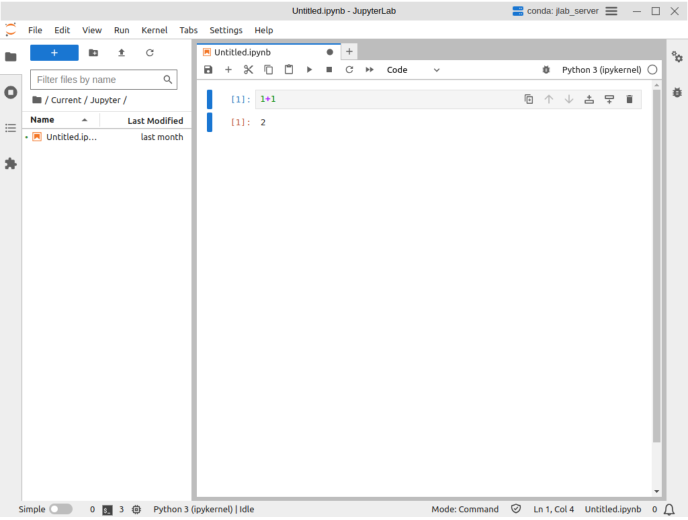
Figure 1: The Interactive Python Shell

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. However, it lacks advanced features found in full-fledged Integrated Development Environments (IDEs), 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 Unbuntu 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.

Jupyter Notebooks Jupyter notebooks offer a more interactive and versatile platform, particularly favored in data science and academic research. Jupyter Notebooks allow users to create and share documents that can contain ”cells” where each cell may contain Python code, text (using Markdown), 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



```
[1]: 1+1
[1]: 2
```

Figure 2: Jupyter Notebook

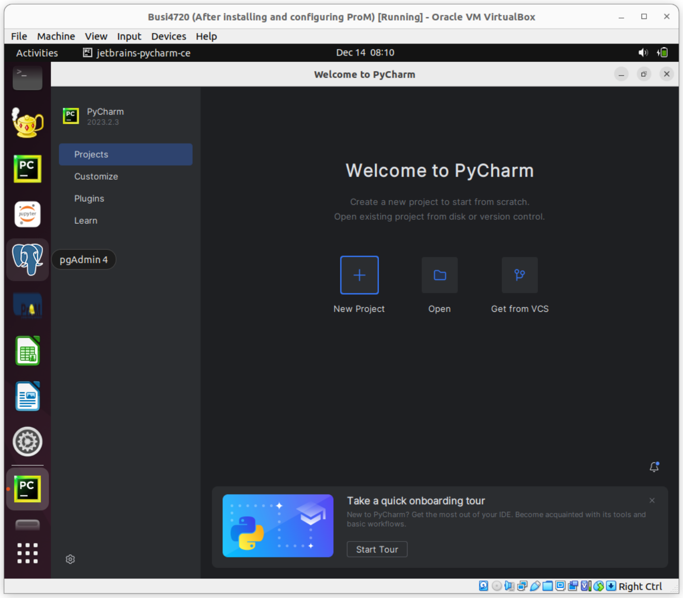
process is as important as the code itself, allowing for a narrative approach to coading. While Jupyter Notebooks support various programming languages, they are predominantly used with Python. Figure 2 shows a screenshot of a Jupyter notebook in the JupyterLabs Desktop environment.

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 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.

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



Welcome to PyCharm

Create a new project to start from scratch.
Open existing project from disk or version control.

Take a quick onboarding tour

New to PyCharm? Get the most out of your IDE. Become acquainted with its tools and basic workflows.

Figure 3: PyCharm Integrated Development Environment (IDE)

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.

4 Python Basics

The basic Python code in the following example prints a character string. 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:

```
print('hello world')

age = 19
name = 'Malina'
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)
```

Multiline character strings are enclosed in triple quotes, and the line breaks form part of the string, as shown in the following example. Note that the `print(s)` function prints the line breaks in the character string.

```
s = '''This is line 1
and here is line 2
and now this is line 3'''
print(s)
```

As with R, you can use Python interactively as a calculator. It provides the usual arithmetic operators and comparison 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:

```
2 + 2
2**4
13 // 3
-13 // 3
13 % 3
-25.5 % 2.25
3 < 5
3 > 5
3 == 5
(3 < 5) and (4 < 2)
(3 < 5) or not (4 < 2)
```

The next code example shows some useful string functions. The `startwith()` 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'
if language.startswith('Innu'):
    print('Yes, the string starts with "Innu"')
if 'u' in language:
    print('Yes, it contains the string "u"')
if language.find('nuk') != -1:
    print('Yes, it contains the string "nuk"')
```

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

```
# Joining and Splitting
delimiter = '_*_'
mylist = ['Nain', 'Hopedale', 'Makkovik', 'Rigolet']
mystring = delimiter.join(mylist)
print(mystring)
thelist = mystring.split(delimiter)
print(thelist)
```

> **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.

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.

```
# Inuit deities
gods = ['Sedna', 'Nanook', 'Akna', 'Pinga']
print('There are', len(gods), 'deities:')
for god in gods:
    print(god, end=' ')

# Appending to a list
gods.append('Amaguq')
print('\nThe list of deities is now', gods)

# Sorting a list
gods.sort()
print('The sorted list is', gods)

# Removing items from a list
print('The first deity is', gods[0])
olditem = gods[0]
del gods[0]
print('I removed', olditem)
print('The list is now', gods)
gods.remove('Pinga')
print('The list is now', gods)
```

The above example also shows the use of Python comments, beginning with `#` to the end of the line. The example also shows iteration (”repeating”) 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 Phython are also ordered collections of items, but they are immutable, i.e. their contents cannot be changed. Tuples use round brackets `()` as delimiters.

```
# Inuit Nunangat
regions = ('Inuvialuit', 'Nunavut', 'Nunavik', 'Nunatsiavut')
print('Number of regions is', len(regions))

all_regions = 'NunatuKavummiut', 'Kalaallit', 'Inupiaq', regions
print('Number of all Inuit regions:',len(all_regions))
print('All Inuit regions are', all_regions)
print('Regions in Inuit Nunangat are', all_regions[3])
print('First region in Inuit Nunangat is', all_regions[3][1])
print('Number of all Inuit regions is', \
    len(all_regions)-1+len(all_regions[3]))
```

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.

```python
# Largest citites
c = {
    'Inuvialuit': 'Inuvik',
    'Nunavut': 'Iqaluit',
    'Nunavik': 'Kuujjuaq',
    'Nunatsiavut': 'Nain'
}
print("Nunavik's largest city is", c['Nunavik'])
```

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.

```python
# Retrieving keys and values
print(list(c.keys()))
print(list(c.values()))

# Deleting a key-value pair
del c['Nunavut']
print('\nThere are {} cities left\n'.format(len(c)))
for region, city in c.items():
    print('{} is largest city of {}'.format(city, region))

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

```python
towns = ['Hopedale', 'Makkovik', 'Nain', 'Postville', 'Rigolet']
pops = [596, 365, 1204, 188, 327]
pop_by_town = dict(zip(towns, pops))
print(pop_by_town)
```

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.

```
regions = ('Inuvialuit', 'Nunavut',
           'Nunavik', 'Nunatsiavut')
# Slicing on a tuple
print('Item 1 to 3 is', regions[1:3])
print('Item 2 to end is', regions[2:])
print('Item 1 to -1 is', regions[1:-1])
print('Item start to end is', regions[:])
```

Slicing in Python is more advanced than slicing in R as not only the beginning and end index can be specified, but also the step size, as shown in the next Python code block. The final example slices backwards.

```
# Slicing with step
print(regions[::1])
print(regions[::2])
print(regions[::3])
print(regions[::-1])
```

Character strings are also sequences, and they support slicing or indexing the same way as other sequences in Python.

```
language = 'Innuktitut'
# Slicing on a string
print('characters 1 to 3 is', language[1:3])
print('characters 2 to end is', language[2:])
print('characters 1 to -1 is', language[1:-1])
print('characters start to end is', language[:])
```

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.

Tip: To read and execute Python statements from a file, use the expression `exec(open('filename.py').read())`

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.

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 powerful 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.

NumPy arrays facilitate advanced mathematical and statistical operations, including linear algebra, Fourier transform, and random number generation. The ndarray object supports vectorized operations, broadcasting, and indexing capabilities. This means that operations can be applied to entire arrays without the need for explicit loops, leading to cleaner and faster code.

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 scien-

`ndarray.ndim`	Number of axes
`ndarray.shape`	Typle describing the size of each axis (dimension)
`ndarray.size`	Total number of elements
`ndarray.dtype`	The datatype of the elements, for example `numpy.int32`, `numpy.int16`, `numpy.float32`, or `numpy.float64`
`ndarray.itemsize`	Number of bytes for each element

Table 1: Attributes of NumPy ndarray

tific computing all build upon and work in conjunction with NumPy, creating a robust ecosystem for scientific computing tasks.

Tip: The NumPy website provides two very good introductions, in the form of the Quick Start and the NumPy for absolute beginners tutorials.

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 `reshape`d 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)

print(a.shape)
print(a.ndim)
print(a.dtype.name)
print(a.size)
print(type(a))
```

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` parrameter in the `max` function to specify whether to aggregate by row or by column. 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, 5, 6)])

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

# Array operations
print(np.max(b))
print(np.max(b, axis=0))
print(np.max(b, axis=1))
print(np.std(b))
print(np.cov(b))
print(np.sum(b))
```

In the above example, the `std()` function without `axis` parameter computes the standard deviation of all elements in the array, while `cov` treats each *row* of the array as a vector and computes their variances and covariances. To treat array columns as vectors, either transpose the array first, using the `T` operator or use the `rowvar=False` parameter for the `cov()` function.

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 dimensions 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]])
print(b[2, 3])
print(b[0:5, 1])
print(b[:, 1])
print(b[1:3, :])
print(b[-1])
```

When not all axes are supposed to be sliced, one can omit initial or final unsliced axes in the slicing expression using the ellipsis ”...” as shown in the following Python code block.

```
c = np.array([[[  0,   1,   2],
               [ 10,  12,  13]],
              [[100, 101, 102],
               [110, 112, 113]]])
print(c[1, ...])
print(c[1, : , : ])
print(c[..., 2])
print(c[: , : , 2])
print(c[..., : , 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[2] (rng) in NumPy to create an array of shape `(3, 4)` filled with random numbers between 0 and 1.

```
# Create a random number generator with seed 1
rg = np.random.default_rng(1)

# Create an array of shape (3, 4) of random numbers
a = np.floor(10 * rg.random((3, 4)))

# Show information about the array and reshape
print(a.shape)
print(a.flatten())
print(a.reshape(6, 2))
print(a.T)
print(a.T.shape)
```

[2] A random number generator in computer science is always a pseudo-random number generator that 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.

The above example uses the flatten() function which returns a one-dimensional array, whereas the flat property returns an iterator to be used in a for loop. The T property returns the transpose of the array. In two dimensions, the transpose swaps rows and columns. The NumPy transpose is also defined for more than two dimensions, the axes are transposed such that a.T.shape==a.shape[::-1].

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.

```
b = np.floor(5 * rg.random((3, 4)))
print(np.vstack((a, b)))
print(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]])
print(a[a < 5])
print(a < 5)
print(a[a%2 == 0])
print(a%2 == 0)
```

Finally, NumPy provides easy ways to identify unique elements in an array and to count how often particular elements occur in an array. The following example also demonstrates another use of the zip() function, already introduced above, to construct a list of tuples.

```
a = np.array([11, 11, 12, 13, 14, 15, 16,
              17, 12, 13, 11, 14, 18, 19, 20])
print(np.unique(a))

# Return the first index of a unique value
values, indices = np.unique(a, return_index=True)
print(list(zip(values, indices)))

# Return the counts of each unique value
values, counts = np.unique(a, return_counts=True)
print(list(zip(values, counts)))
```

Hands-On Exercises

1. Create an 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

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.

Tip: The Pandas website provides very good 10 Minutes to Pandas introductory tutorial for Pandas.

Central to Pandas are two primary data structures: the DataFrame and the Series. 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 example also shows how a Python dict can be converted into a series with named elements.

```python
# 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 and reorder the entries
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 contains no value for the key ”d”.

Pandas series behave largely like NumPy arrays, but note that 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`.

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

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

# Series have a datatype and name
s.name = 'My First Series'
print(s.dtype)
```

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.

```
d = {
    "one": pd.Series([1.0, 2.0, 3.0],
                index=['a', 'b', 'c']),
    "two": pd.Series([1.0, 2.0, 3.0, 4.0],
                index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)
print(df.index)
print(df.columns)
print(pd.DataFrame(d, index=['d', 'b', 'a'],
                      columns=['two', 'three']))
```

Constructing the DataFrame `df` ”lines up” the two Series on their common indices, and will introduce a ”NaN” for index ”d” in column ”one”, because that Series does not contain a value for ”d”. Similarly, inserting a column named ”three” in the last line of the above example yields a column filled with ”NaN” because the dict `d` does not contain values for the key ”three”.

DataFrame columns can be accessed using their quoted name, and will yield a Pandas Series with the usual operations. The following Python code example shows that new columns can be added simply by defining them, as in the ”flag” column below or using the `assign()` function, which works similarly to the `mutate` function in R/dplyr. Columns can be removed using the `del` command or the `pop()` function. The latter returns the deleted column as a Series.

```
print(df['one'])
df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2
print(df)

del df['two']
three = df.pop('three')
df['foo'] = 'bar'
df['one_trunc'] = df['one'][:2]
df.insert(1, 'bar', df['one'])
print(df)

# Similar to 'mutate' in R/Dplyr
df = df.assign(four = df['one'] * np.sqrt(df['bar']))
print(df)
```

Pandas DataFrames can be index by colum, by label, by integer location, or by boolean vectors. Table2 shows an overview of the different methods and their return values.

Select column	`df[’colname’]`	Series
Select row by label	`df.loc[’label’]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[::]`	DataFrame
Select rows by boolean vector	`df[bool]`	DataFrame

Table 2: Methods for indexing Pandas DataFrames

As noted earlier, Pandas automatically aligns data by indices, that is, by row and column labels, for operations on dataframes. Note how the addition of two dataframes of unequal shape introduces ”NaNs”. 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.

```
df = pd.DataFrame(np.random.randn(10, 4),
                  columns=["A", "B", "C", "D"])
df2 = pd.DataFrame(np.random.randn(7, 3),
                   columns=["A", "B", "C"])
print(df + df2)

# Elementwise operators
print(df * 5 + 2)
print(1/df)
print(df**4)

# Transpose
print(df.T)

# Using Numpy functions
print(np.exp(df))
print(np.asarray(df))
```

To apply element-wise character string operations on Series or DataFrames it is useful to use the `str` property:

```
# String functions with 'str'
s = pd.Series(
    ["A", "B", "C", "Aaba", "Baca", np.nan,
     "CABA", "dog", "cat"], dtype="string")
s.str.lower()
```

Pandas provides a number of useful functions to get information about the contents of a DataFrame. 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.

```
df.info()
df.head()
df.tail(3)
```

The *boolean reduction* functions `all()` and `any()` operate by column on DataFrames with boolean values. As their names suggest, `all()` returns `True` when the all entries in a column are true, whereas `any()` returns `True` if any of the entries in a column are true. The last line of the following Python code block re-applies `any()` to the Series that results from the first application of `any()`.

```
# Boolean reductions
(df > 0).all()
(df > 0).any()
(df > 0).any().any()
```

When making comparisons on DataFrames that include ”NaN”, it is important to realize that two ”NaNs” are not equal when using the == operator, but they are equal when using the `equals` function. The following example illustrates this difference.

```
# NaN's are not the same
df.iloc[0,0] = np.nan
(df+df == df*2).all()
(df + df).equals(df*2)
```

Pandas provides useful functions for basic descriptive statistics and aggregation on DataFrames. In particular, the `describe()` function is useful to get a basic information on the data in a DataFrame. The `mean()` function takes as its optional first argument the axis number (0 for rows, 1 for columns) 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
df.mean(0)
df.mean(1, skipna=False)
df_std = (df - df.mean()) / df.std()
df.describe()

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

Pandas DataFrames 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 by values
df.sort_values(by=['A', 'B'])
df.nsmallest(3, 'A')
df.nlargest(3, 'A')
```

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=list('abc'))

# 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')
```

The `query()` function can also be used for membership tests in Series and DataFrames using the `in` operator. This also is much more compact and easy to read than the pure Python `isin()` function. The following example code block shows the pure Python selection followed by equivalent selection with `query()`:

```
df = pd.DataFrame({'a': list('aabbccddeeff'),
                   'b': list('aaaabbbbcccc'),
                   'c': np.random.randint(5, size=12),
                   'd': np.random.randint(9, size=12)})

# Pure Python versus Query
df[df['a'].isin(df['b'])]
df.query('a in b')

df[~df['a'].isin(df['b'])]
df.query('a not in b')

df[df['b'].isin(df['a']) & (df['c'] < df['d'])]
df.query('a in b and c < d')

df[df['b'].isin(["a", "b", "c"])]
df.query('b == ["a", "b", "c"]')

df[df['c'].isin([1, 2])]
df.query('[1, 2] in c')
```

Pandas DataFrames also offer easy functions to remove duplicates. The following

Python example code block shows how to identify rows that contain duplicate elements in a list of columns, and then remove the duplicates, keeping either the first or the last row. Note the different row indices in the retained results of `drop_duplicates` and their different values columns ”c” and ”d”.

```
df2 = df.copy()

df2.duplicated(['a', 'b'])
df2.drop_duplicates(['a', 'b'], keep='last')
df2.drop_duplicates(['a', 'b'], keep='first')
```

Finally, Pandas provides many functions for reading and writing DataFrames from and to a variety or serialization formats and even SQL RDBMS. See the Pandas IO user guide for details.

7 The Pagila Database in Pandas

This section the use of Pandas for descriptive data analysis using the Pagila database data as an example. The Pagila database[3] is a demonstration database originally developed for teaching and development of the MySQL RDBMS under the name Sakila[4]. Pagila is designed as a sample database to illustrate database concepts and is based 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.

[3] `https://github.com/devrimgunduz/pagila`, `https://github.com/devrimgunduz/pagila/blob/master/LICENSE.txt`

[4] `https://dev.mysql.com/doc/sakila/en/`, `https://dev.mysql.com/doc/sakila/en/sakila-license.html`

```python
# Read CSV
rentals = pd.read_csv('rentals.csv')

# Convert data types
rentals['rental_date'] = \
    pd.to_datetime(rentals['rental_date'], utc=True)
rentals['return_date'] = \
    pd.to_datetime(rentals['return_date'], utc=True)
rentals['payment_date'] = \
    pd.to_datetime(rentals['payment_date'], utc=True)

# Basic information
rentals.info()
rentals.describe()
rentals.index
rentals.columns
rentals.shape
```

When working with data, it is often useful to first identify and remove missing values. The following Python code block first identifies columns (`axis=1`) in the dataset that contain any (`any()`) missing values (`isna()`). Of these filtered rentals, only some columns are selected.

```python
filtered_rentals = rentals[rentals.isna().any(axis=1)]

selected_rentals = \
    filtered_rentals[
      ['last_name', 'rental_date', 'return_date', 'title', 'amount']]
```

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.

```python
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.

```
actors = pd.read_csv('actors.categories.csv')

result = pd.merge(rentals, actors, on='title',
             suffixes=('_customer', '_actor'), how='outer')

result = result[result['rating'] == 'PG']
result['actor'] = result['last_name_actor'] + \
                   ', ' + result['first_name_actor']

result.rename(columns={'release_year': 'year'}, inplace=True)

result = result[['actor', 'title', 'category', 'year']]

result.drop_duplicates(['actor', 'title', 'category', 'year'],
          inplace=True)

result.sort_values(['category', 'year', 'title'], inplace=True)

grouped = result.groupby(['category', 'year', 'title'])

g_result = grouped['actor'].apply(list).reset_index()

print(g_result)
```

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.

- *Reading Data*: The code reads a CSV file named 'actors.categories.csv' into a DataFrame called "actors".
- *Merging DataFrames*: It then merges two DataFrames: "rentals" and "actors", based on the "title" column that is common to both DataFrames. The `suffixes` parameter is used to differentiate columns with the same name in both DataFrames, by adding either "_customer" or "_actor" to the column names. The `how='outer'` parameter ensures that all records from both DataFrames are included in the result, even if there are no matching titles in one of them.
- *Filtering Data*: After merging, the script filters the resulting DataFrame to include only rows where the "rating" column contains the value "PG".
- *Creating a New Column*: A new column, "actor", is created 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".
- *Selecting and Rearranging Columns*: The DataFrame 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.

- *Sorting Data*: The DataFrame is sorted by ”category”, then ”year”, and finally ”title”.
- *Grouping Data and Creating a List*: The data is grouped by ”category”, ”year”, and ”title”. 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 DataFrame.
- *Printing the Final Result*: Finally, the processed DataFrame is printed.

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 merges the ”rentals” DataFrame with the ”addresses” DataFrame based on the columns ”customer_address” in ”rentals” and ”address_id” in ”addresses” to linking rentals with corresponding customer addresses.
- The script then merges the resulting DataFrame with the ”actors”, based on the ”title” column.

```
addresses = pd.read_csv('addresses.csv')
addresses['phone'] = addresses['phone'].astype(str)

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 required analysis to on the full data constructed above, using the following steps:

- The code groups the data by ”city” and ”actor” and calculates the size of each group. This results in a count of how many times each actor’s movies were rented in each city. The result is reset into a DataFrame ”grouped” with a new column ”count” representing these sizes.
- Within each city, actors are ranked based on the ”count” column, with the ranking stored in a new column ”ranking”. 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 DataFrame to select the top 3 actors (or ties) in terms of rental counts in each city.

- The filtered data is then sorted by ”city”, ”ranking”, and ”actor” before being printed.

```
full_data['actor'] = full_data['last_name_actor'] + ', ' + \
                     full_data['first_name_actor']

grouped = full_data.groupby(['city', 'actor']).size() \
            .reset_index(name='count')

grouped['ranking'] = grouped.groupby('city')['count'] \
            .rank(method='min', ascending=False)

filtered = grouped[grouped['ranking'] < 4]

sorted_df = filtered.sort_values(by=['city', 'ranking', 'actor'])

print(sorted_df)
```

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['customer'] = full_data['first_name_customer'] + ' ' + \
                        full_data['last_name_customer']

selected_data = full_data[['customer', 'amount', 'rental_duration', \
                           'category', 'phone', 'city']]

grouped_data = selected_data \
    .groupby(['category', 'rental_duration', 'customer']) \
    .agg(payments=('amount', 'sum'), num_rentals=('amount', 'count')) \
    .reset_index()

grouped_data['ranking'] = grouped_data \
      .groupby(['category', 'rental_duration'])['payments'] \
      .rank(method='min', ascending=False)

top_entries = grouped_data.loc[
    grouped_data.groupby(['category', 'rental_duration'])['ranking'] \
                .idxmin() ]

print(top_entries)
```

By now, it should be clear what most of the functions in the analysis accomplish. However, two important new things to note. First, the `agg()` function computes aggregates of the values in its first argument using the function in its second argument and stores the aggregate values in a new column. For example, the code below creates a new column ”payments” with the ”sum” of the values of the ”amount” column of the grouped data, and a new column ”num_rentals” with the ”count” of the values of the ”amount” column. Second, 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['customer'] = full_data['first_name_customer'] + ' ' + \
                        full_data['last_name_customer']

full_data['q'] = pd.to_datetime(full_data['rental_date']).dt \
                   .to_period("Q")

selected_data = full_data[['customer', 'q', 'amount', 'rental_date']]

grouped_data = selected_data.groupby(['q', 'customer']) \
        .agg(payments=('amount', 'sum')).reset_index()

distinct_data = grouped_data \
        .drop_duplicates(['customer', 'q', 'payments'])

distinct_data['rank_top'] = distinct_data \
        .groupby('q')['payments'].rank(method='min', ascending=False)

distinct_data['rank_bot'] = distinct_data \
        .groupby('q')['payments'].rank(method='min', ascending=True)

filtered_data = distinct_data[(distinct_data['rank_top'] < 6) |
                              (distinct_data['rank_bot'] < 6)]

sorted_data = filtered_data \
        .sort_values(by=['q', 'payments'], ascending=[True, False])

print(sorted_data)
```

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

The code below introduces *Lambda* functions. Lambda functions are unnamed, in-line functions, here it is a function that converts its parameter `x` to a set (i.e. it removes duplicates), and then converts the set to a list. The Lambda function is used as an argument to the `apply` function, that is, it is applied to all elements of the ”titles” column in the grouped data. Recall that the ”titles” column was introduced earlier in the script when the `list` function was applied to the ”title” column of the grouped data and contains a list of film titles.

```python
full_data['customer'] = \
    full_data['first_name_customer'] + ' ' + \
    full_data['last_name_customer']

selected_data = full_data[['customer', 'title']]

grouped_data = selected_data \
        .groupby('customer')['title'] \
        .apply(list) \
        .reset_index(name='titles')

grouped_data['rentals'] = grouped_data['titles'].apply(len)

grouped_data['unique_titles'] = grouped_data['titles'] \
        .apply(lambda x: list(set(x)))

grouped_data = grouped_data.drop(columns=['titles'])
sorted_data = grouped_data.sort_values(by='customer')

print(sorted_data)
```

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