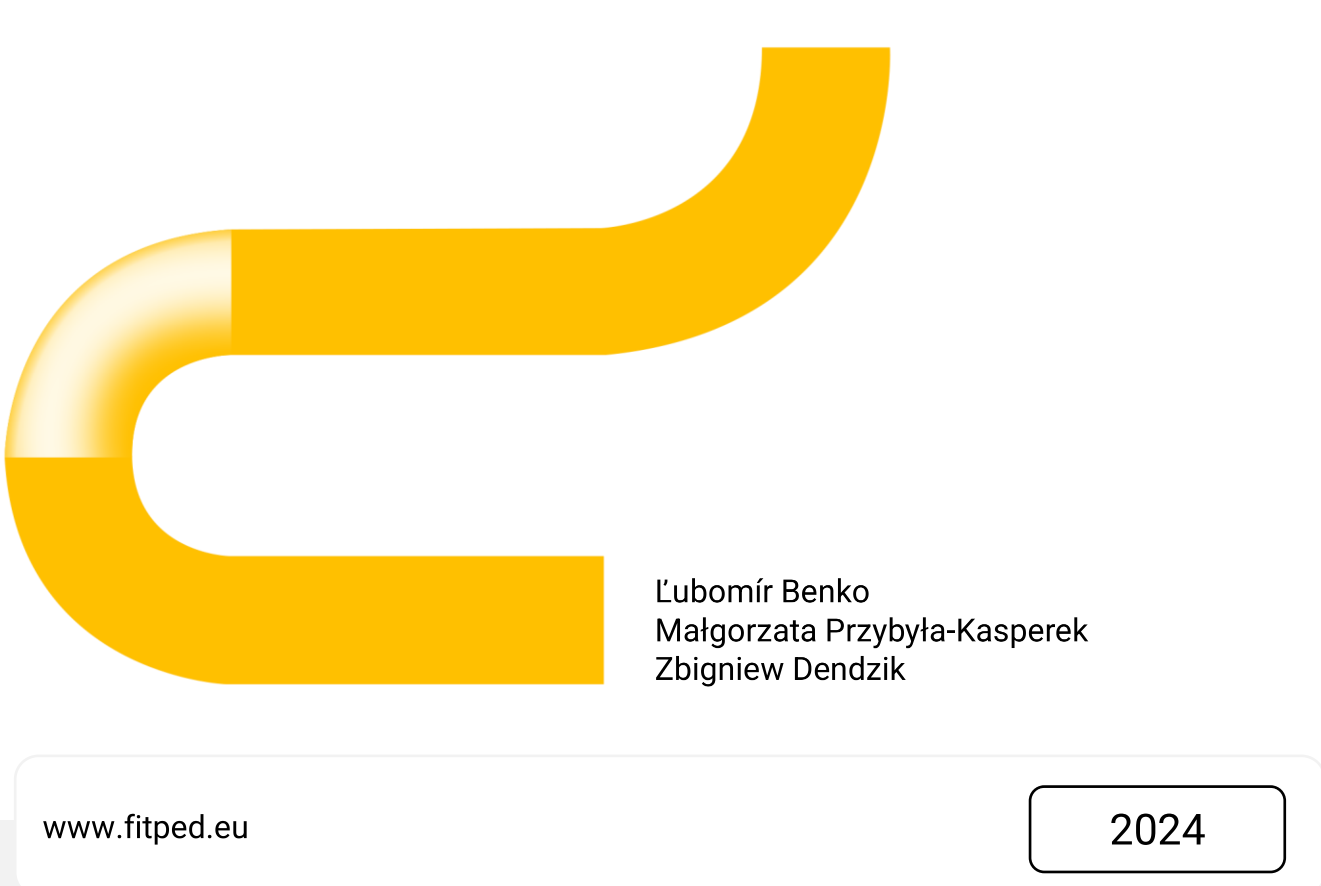
Data Pre-processing Techniques



Ľubomír Benko
Małgorzata Przybyła-Kasperek
Zbigniew Dendzik

www.fitped.eu

2024

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

Data Pre-processing Techniques

Published on

November 2024

Authors

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Zbigniew Dendzik | University of Silesia in Katowice, Poland

Reviewers

Piet Kommers | Helix5, Netherland

Roman Valovič | Mendel University in Brno, Czech Republic

Cyril Klimeš | Mendel University in Brno, Czech Republic

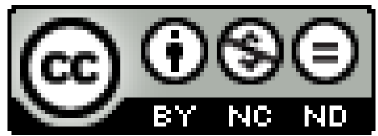
Vladimiras Dolgopolovas | Vilnius University, Lithuania

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.



Licence (licence type: Attribution-Non-commercial-No Derivative Works) and may be used by third parties as long as licensing conditions are observed. Any materials published under the terms of a CC Licence are clearly identified as such.

All trademarks and brand names mentioned in this publication and all trademarks and brand names mentioned that may be the intellectual property of third parties are unconditionally subject to the provisions contained within the relevant law governing trademarks and other related signs. The mere mention of a trademark or brand name does not imply that such a trademark or brand name is not protected by the rights of third parties.

© 2024 Constantine the Philosopher University in Nitra

ISBN 978-80-558-2225-9

TABLE OF CONTENTS

1 NumPy Library – Arrays ... 5
  1.1 Numpy ... 6
2 Lambda Expressions ... 14
  2.1 Lambda expressions ... 15
  2.2 Filter and sort ... 19
3 Regular Expressions ... 26
  3.1 Regular expression ... 27
  3.2 Special sequences, character sets ... 32
4 Random Number Generation ... 41
  4.1 Library random - generating random numbers ... 42
5 Pandas ... 45
  5.1 Pandas introduction ... 46
  5.2 Data input ... 48
  5.3 Checking the data ... 52
6 Data Manipulation ... 55
  6.1 Missing values ... 56
  6.2 Data selection ... 62
  6.3 Adding a new column into DataFrame ... 72
  6.4 Removing data from DataFrame ... 74
  6.5 Working with data in DataFrame ... 80
7 Data Summarization ... 84
  7.1 Data grouping ... 85
  7.2 Pivot tables ... 89
  7.3 Visualization ... 95
  7.4 Category variables ... 103
8 Project ... 112
  8.1 Spaceship Titanic - basic characteristics ... 113
  8.2 Spaceship Titanic - working with data ... 119

NumPy Library – Arrays

Chapter 1

1.1 Numpy

1.1.1

Numpy arrays

Numpy operates very efficiently on numeric data arranged in arrays.

```
import numpy as np
a = np.array([1.1, 3., 3.3, 7., 11.1, 12.3, 15.])
b = np.array([1.0, 3.7, 4.6, 7.3, 11., 12., 15.3])

print(a)
print(b)
```

1.1.2

Data types

All data stored in variables and, in particular, in Numpy arrays are assigned a specific type. The type may be inferred from the assignment

```
print( a.dtype )

k = np.array([1, 3, 4, 7, 1])
print( k.dtype )

k = np.array([1.1, 3.5, 4.3, 7.8, 1.])
print( k.dtype )
```

... or may be directly specified while creating the array.

```
k = np.array([1.1, 3.5, 4, 7, 1], dtype=int)
print( k.dtype )
```

1.1.3

Generating arrays

Numpy arrays may also be generated using functions available in Numpy.

```
x = np.linspace(0.1, 10*np.pi, 500)
print(x)
```

1.1.4

Operating on Numpy arrays

Operations on Numpy arrays are elementwise.

```
m = 3 * a
print(m)
```

Program output:

```
[ 3.3  9.   9.9 21.  33.3 36.9 45. ]
```

```
s = a + b
print(s)
```

Program output:

```
[ 2.1  6.7  7.9 14.3 22.1 24.3 30.3]
```

```
t = np.cos(a)
print(t)
```

Program output:

```
[ 0.45359612 -0.9899925  -0.98747977  0.75390225  0.10423603
0.96473262
 -0.75968791]
```

```
r = a < b
print(r)
```

Program output:

```
[False  True  True  True False False  True]
```

1.1.5

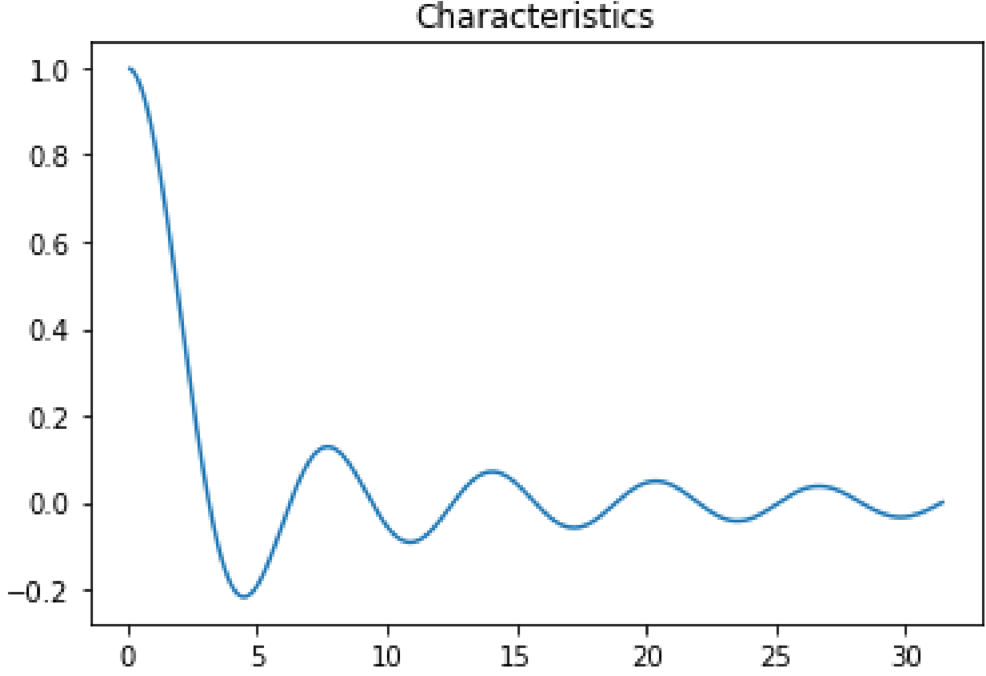
Visualizing data stored in Numpy arrays

An convenient way to visualize data stored in Numpy arrays is to use Matplotlib.

```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0.1, 10*np.pi, 500)
plt.plot(x, np.sin(x)/x)
plt.title('Characteristics')
plt.show()
```

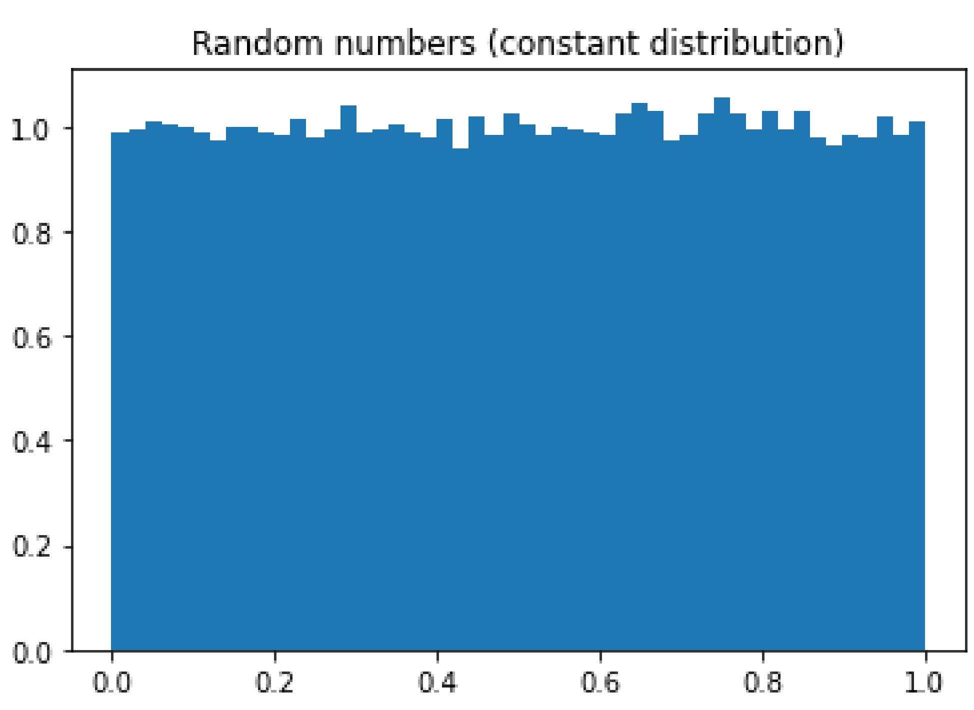
Program output:




```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

v = np.random.ranf(100000)
plt.hist(v, bins=50, density=1)
plt.title('Random numbers (constant distribution)')
plt.show()
```

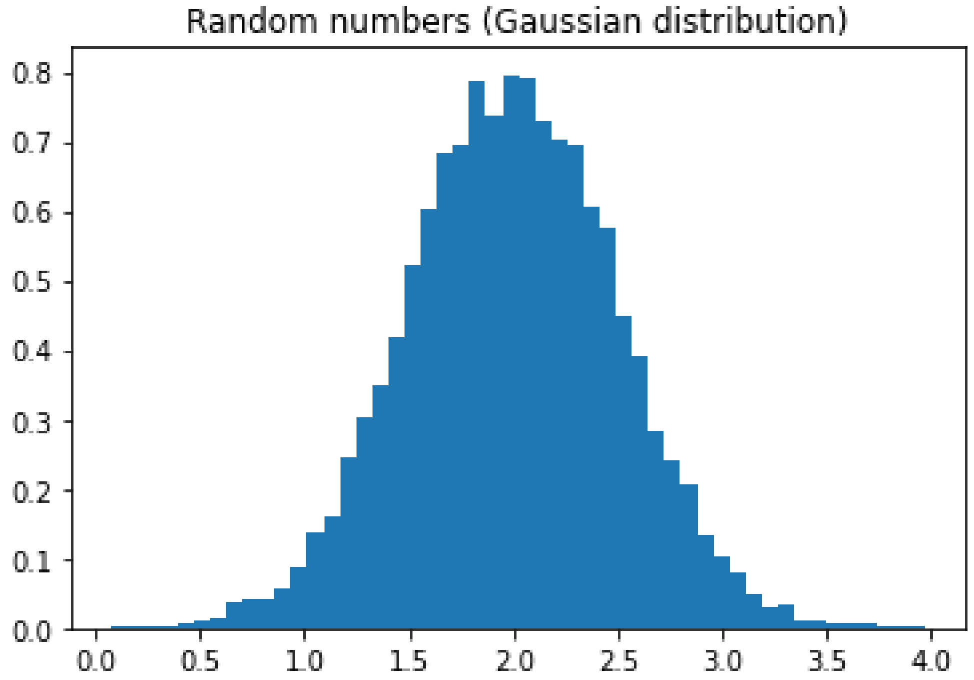
Program output:



```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

mean,sdev = 2,0.5
rg = np.random.default_rng(1)
v = rg.normal(mean,sdev,10000)
plt.hist(v, bins=50, density=1)
plt.title('Random numbers (Gaussian distribution)')
plt.show()
```

Program output:



1.1.6

Task

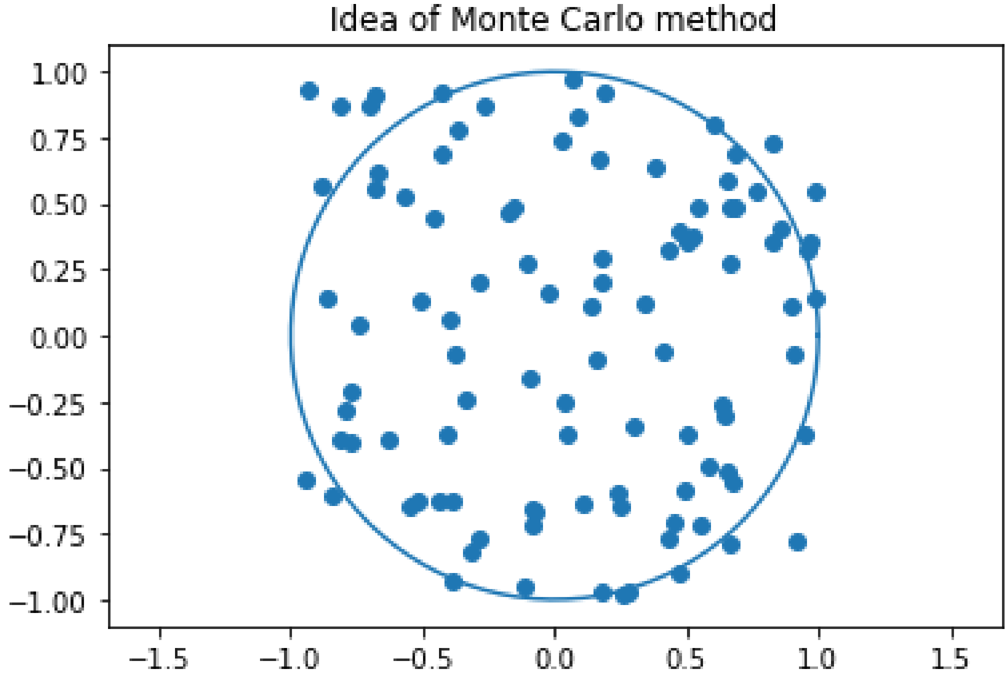
Evaluate the value of pi using popular approach based on Monte Carlo method for n=10000 trials. This method is based on counting random hits inside the whole square with side 1 and hits inside the quarter of a circle with radius 1 and centre in one of the corners of this square. Below you find graphical illustration of this concept. An answer to this task is the calculated value of pi, up to six decimal places.

As a follow up, you may perform calculations and generate the plot of standard deviation as a function of number of Monte Carlo trials.

```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0,2*np.pi,100)
plt.plot(np.cos(t),np.sin(t))
x,y = 2*np.random.rand(2,100) - 1
plt.scatter(x,y)
plt.axis('equal')
plt.title('Idea of Monte Carlo method')
plt.show()
```

Program output:



1.1.7

Numpy arrays - further examples

Importing Numpy

```
# first import numpy
import numpy as np
```

1. Task: create a NumPy array using the arange() function with numbers from interval -10 to 10

```
# create the array from interval and print it
```

```
ar = np.arange(-10,11) # if we want to include the 10, we need
to add 1 number more (because of zero)
print(ar)
```

Program output:

```
[-10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4
5   6   7
   8   9  10]
```

2. Task: create a NumPy array of 100 zeros with the zeros() function

```
# create the array of 100 zeros and print it (we can add the
parameter dtype=int to have integers)
ar_zeros = np.zeros((100,), dtype=int)
print(ar_zeros)
```

Program output:

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

3. Task: create two random NumPy arrays with 10 numbers (use function random.randint(low=0, high=100,...) and print the addition of both arrays

```
# create the first array arr1
arr1 = np.random.randint(low=0, high=100, size=10)
# create the second array arr2
arr2 = np.random.randint(low=0, high=100, size=10)
print(arr1+arr2)
```

Program output:

```
[ 94 155  73 147 113  66 114 132  63 135]
```

1.1.8

Task 1

Two numpy array are given. Your task is to print the digit sum of the highest number and what array contains the number.

Example of the output:

125, 2. array

```
import numpy as np
```

```
arr1 =
np.array([275,44,6100,871,1022,485241,2237,477000,22113,4849,1
2374,0])
arr2 =
np.array([1,841,389163,158,31530,800000,151230,12348,231,3889,
3107])

# write your solution here
digmax = -1
index = 0
for d in arr1:
  digsum = 0
  while d//10>0:
    digsum += d%10
    d = d//10
  if digsum>digmax:
    digmax = digsum
    index = 1

for d in arr2:
  digsum = 0
  while d//10>0:
    digsum += d%10
    d = d//10
  if digsum>digmax:
    digmax = digsum
    index = 2
print(str(digmax)+", "+str(index)+". array")
```

Program output:

```

```

1.1.9

Task 2

```
Multiple numpy matrices are given. First run the code below
which changes the values of the three matrices. Your task is
to identify the matrix that has the lowest average.
import numpy as np
matrix1 = np.full((3,3),8)
matrix2 = np.array([[10,11,12],[4,5,6],[7,8,9]])
matrix3 = np.array([[11,72,-3],[20,0,18],[-1,-3,-7]])
matrix4 = np.multiply(matrix2, matrix1)
matrix5 = np.subtract(matrix3, matrxi2)
```

Program output:

```
```

```
# here write your code
avg1 = np.average(matrix1)
avg2 = np.average(matrix2)
avg3 = np.average(matrix3)
avg4 = np.average(matrix4)
avg5 = np.average(matrix5)
averages = np.array([avg1,avg2,avg3,avg4,avg5])
print(np.argmin(averages, axis=0)+1)
print(averages)
```

Program output:

- matrix1
- matrix2
- matrix3
- matrix4
- matrix5

Lambda Expressions

Chapter 2

2.1 Lambda expressions

2.1.1

In Python, we can often simplify code by using Lambda functions. Lambda is a function without a name which means it is an anonymous function.

The characteristics of this function are as follows:

- The function can have multiple arguments, separated by commas
- The function can consist of only one line of code. It is restricted to a single expression.
- It is good for short operations/data manipulations.

An example of Lambda function can be found below.

```
multi = lambda x, y : x * y
print(multi(3,4))
print(multi("Hello",4))
```

Program output:

```
12
HelloHelloHelloHello
```

The above function can be also written as follows

```
def multi(x,y):
    return x * y
print(multi(3,4))
print(multi("Hello",4))
```

Program output:

```
12
HelloHelloHelloHello
```

The result and meaning are exactly the same, but the code is more concise.

2.1.2

Mark all true statements about the lambda function

- A lambda function can have at most one line of code.
- A lambda function can have at most two lines of code
- The lambda function is anonymous
- The default name of the lambda function is lambda.

2.1.3

Python Lambda Function can be used with list comprehension. List comprehension is a simple way to create a new list based on the values of an existing list.

The general pattern is as follows:

newlist = [*expression* **for** *item* **in** *iterable* **if** *condition == True*]

An example of the list is given below.

```
names = ["David", "Philip", "Jan", "Tom", "Mark"]
ComprehensionList = [x for x in names if "a" in x]
print(ComprehensionList)
```

Program output:

```
['David', 'Jan', 'Mark']
```

```
#We want to determine triples from the set {1,...,10}
satisfying the condition x^2+y^2<=z^2
TripleCondition=[(x,y,z) for x in range(1,10) for y in
range(1,10) for z in range(1,10) if x**2 + y**2 <= z**2]
print(TripleCondition)
```

Now we can use the lambda function in conjunction with list comprehension, which significantly extends the range of possibilities.

```
names = ["David", "Philip", "Jan", "Tom", "Mark"]
numbers = [1,2,3,4,5]
multi = [lambda arg1=x,arg2=y: arg1*arg2 for x in names for y
in numbers]

for item in multi:
    print(item())
```

The above code works as follows:

Inside the list comprehension, each iteration creates a new lambda function with a default argument of x (where x is the current item in the iteration). Inside the for loop, the same function is calling with object having the default argument using item().

2.1.4

Please select the description of the purpose of the lambda function

```
first = [2,3]
second = [5,3]
```

```
third = [2,9]
multi = [lambda arg1=x,arg2=y,arg3=z: "Yes" if ((arg1+arg2<=""
pre="">
```

```
• The function checks whether a triangle can be constructed
  from three segments of the given length. From three
  lists, it checks all possible combinations of elements
  (one element from one list). It returns the answer Yes if
  a triangle can be built, No otherwise.
• The function checks whether a triangle can be constructed
  from three segments of the given length. It checks
  combinations of segments from one list (three elements
  from one list). It returns the answer yes if a triangle
  can be built, no otherwise.
• The function checks whether a triangle can be constructed
  from three segments of the given length. It checks
  combinations of only the first element from each list.
  Returns the answer yes if a triangle can be built, no
  otherwise.
```

2.1.5

As we know, the lambda function does not allow using multiple lines of code. However, this can be overcome by using several lambda functions. For example, the following code will return the median of the elements in the list. Note in the above code the function *sorted* is used, which will be explained in the next lessons - it returns a sorted list.

```
List = [[1, 4, 2, 5], [3, 6, 1, 8, 19], [5, 7, 2, 8, 10, 2]]
# Sort each sublist
sortList = lambda x: (sorted(i) for i in x)
# Get the median element
median = lambda x, other_lambda : [y[int(len(y)/2)] if (len(y)
% 2 ==1) else (y[int(len(y)/2)-1]+y[int(len(y)/2)])/2 for y in
other_lambda(x)]
results = median(List, sortList)
 print(results)
```

Program output:

```
[3.0, 6, 6.0]
```

2.1.6

In this section, we will have the opportunity to see how useful the lambda function really is. It significantly shortens the code in combination with the other functions presented in this section.

The first important function used in conjunction with the lambda function is the **map** function. It allows us to execute lambda functions on individual elements of a list.

map() is a function which takes two arguments: *function* - is the name of a function that will be used (for us it will be lambda) and the second seq is a sequence for example a list.

result = map(func, seq)

The map function applies the function to all the elements of the sequence seq.

With Python 3, map() function returns an iterator.

```
names = ["David", "Philip", "Jan", "Tom", "Mark"]
WithA = list(map(lambda x: x if "a" in x else "", names))
print(WithA)
```

Another example of code simplification using the map() function is checking whether elements from lists satisfy the inequality

$$x^2 + y^2 \leq z^2$$

Note that the lists do not have to be of equal length.

```
A=[2,3,4,6]
B=[2,3,1]
C=[5,3,8,7,8]
TripleCondition=list(map(lambda x,y,z: "Yes" if x**2 + y**2 <=
z**2 else "No", A,B,C))
print(TripleCondition)
```

2.1.7

Check the code that calculates the volume of a set of spheres with known radii.

- `from math import pi`
- `A=[4, 2.3, 1.5, 4, 5, 4.56]`
- `Volume=list(map(lambda r: 4/3*pi*r**3, A))`
- `from math import pi`
- `A=[4, 2.3, 1.5, 4, 5, 4.56]`
- `Volume=list(map(lambda r: 4/3*pi*r**3 for r in A))`
- `from math import pi`
- `A=[4, 2.3, 1.5, 4, 5, 4.56]`
- `Volume=list(map(A, lambda r: 4/3*pi*r**3))`

2.1.8

Check the result of following code.

```
shopping = [ ["12345", "Milk", 4, 13.45],
             ["45368", "Bread", 2, 56.2],
             ["24794", "Butter", 5, 17.87]]
discount = 0.01
total = list(map(lambda x: (x[1],x[2]*(1-discount)) if x[2] >=
100 else (x[1], x[2]), map(lambda x: (x[0],x[1],x[2] * x[3]),
shopping)))
print(total)
```

- `[('Milk', 53.8), ('Bread', 111.27600000000001), ('Butter', 89.35000000000001)]`
- `[('Milk', 53.8), ('Bread', 112.4000000000001), ('Butter', 89.35000000000001)]`
- `[('12345','Milk', 53.8), ('45368','Bread', 111.27600000000001), ('24794','Butter', 89.35000000000001)]`

2.2 Filter and sort

2.2.1

Another very useful function that is used in conjunction with a lambda expression is the filtering function filter(). It is a clear and elegant way to check whether the elements of a list fulfill a certain condition - more specifically, whether the function called on the elements of the list returns True.

The general syntax is as follows:

filter(function, seq)

where *function* returns a Boolean value, True of False. This function will be applied to every element of the list *seq*. An item will be produced by the iterator result of filter() if item is included in the list seq and if function(item) returns True.

See a simple example below:

```
names = ["David", "Philip", "Jan", "Tom", "Mark"]
WithA = list(filter(lambda x: "a" in x, names))
print(WithA)
```

As you can see, this version is even more concise than when using the map() function, because here we don't even have to use a conditional statement. Of course, which function should be used depends on what the goal of our program is.

The next example uses dictionary for filtering.

```
dictOfAnimals = {'antelope' : 'land', 'dolphin' : 'water',
'elephant' : 'land', 'catfish' : 'water', 'carp' : 'water'}
InWater = dict(filter(lambda x: x[1] == 'water',
dictOfAnimals.items()))
print(InWater)
```

Program output:

```
{'dolphin': 'water', 'catfish': 'water', 'carp': 'water'}
```

It is also possible to apply the filter function using the empty *None* function. Then the filter function returns all

elements that are really a certain value.

```
List = ["Noe", "empty", 0, 1, False, True, "0", 0.33]
SomeValue = list(filter(None, List))
print(SomeValue)
```

It is very interesting to use several conditions in one filter function. See example below

```
Tuple=[(1,'antelope','land'),
      (2,'dolphin','water'),
      (3,'elephant','land'),
      (4,'catfish','water'),
      (5,'carp','water')]

def filter1(t): return t[0]>2
def filter2(t): return 'a' in t[1]
def filter3(t): return t[2]==('water')

filters = (filter1,filter2,filter3)
filtered_list = list(filter( lambda x: all(f(x) for f in
filters), List))
print(filtered_list)
```

Program output:

```
[(4, 'catfish', 'water'), (5, 'carp', 'water')]
```

2.2.2

Please check all true statements.

- The filter function uses a function that returns True or False.
- The filter function can receive several separate lists as arguments. For example, the below code is correct if A, B, and C are three lists.

```
filter(function, A,B,C)
```

- The filter function can be used for dictionary and tuple.
- Using the filter function, we must also use the if else statement.

2.2.3

Check the result of following code.

```
library=[(2020,'Clean Code in Python - Second Edition: Develop
maintainable and efficient code','Mariano Anaya',1,422),
      (2022,'Python 3. The Comprehensive Guide','Johannes
Ernesti, Peter Kaiser',2,1036),
      (2015,'The Foundations of Mathematics','Ian Stewart,
David Tall',2,416),
      (2020,'Mathematics for Machine Learning',' Marc Peter
Deisenroth, A. Aldo Faisal, Cheng Soon Ong',3,398)]

def pages(t): return t[4]<500
def title(t): return 'Python' in t[1]
def year(t): return t[0]>=2020
def authors(t): return t[3]<3

uses = (pages,title,authors)
result = list(filter( lambda x: all(f(x) for f in uses),
library))
print(result)
```

- [(2020, 'Clean Code in Python - Second Edition: Develop maintainable and efficient code', 'Mariano Anaya', 1, 422)]
- [(2020,'Clean Code in Python - Second Edition: Develop maintainable and efficient code','Mariano Anaya',1,422),

  (2022,'Python 3. The Comprehensive Guide','Johannes Ernesti, Peter Kaiser',2,1036),

  (2015,'The Foundations of Mathematics','Ian Stewart, David Tall',2,416),

  (2020,'Mathematics for Machine Learning',' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong',3,398)]

- []

- [(2020,'Clean Code in Python - Second Edition: Develop maintainable and efficient code','Mariano Anaya',1,422),

  (2022,'Python 3. The Comprehensive Guide','Johannes Ernesti, Peter Kaiser',2,1036)]

- [(2022,'Python 3. The Comprehensive Guide','Johannes Ernesti, Peter Kaiser',2,1036)]

2.2.4

The two basic ways to sort a list are:

1. Using the function sorted(), which does not change the original object
2. Using the sort method list.sort(), which sorts the original list without creating a copy

The first way - the function sorted() - can also be applied to the dictionary.

```
print(sorted([4,5,5,2,7,2]))
```

Program output:

```
[2, 2, 4, 5, 5, 7]
```

```
List=[4,5,5,2,7,2]
List.sort()
print(List)
```

Program output:

```
[2, 2, 4, 5, 5, 7]
```

```
print(sorted([(1,4),(2,5),(3,5),(4,2),(5,7),(6,2)]))
```

Program output:

```
[(1, 4), (2, 5), (3, 5), (4, 2), (5, 7), (6, 2)]
```

Both the function sorted() and the sort() method have a parameter *key* that specifies the value by which the elements of the list or dictionary should be sorted. In the case of a dictionary, the lambda function is used for defining this parameter. We can specify the variable according to which the elements should be sorted.

```
print(sorted([(1,4),(2,5),(3,5),(4,2),(5,7),(6,2)],key=lambda
x:x[1]))
```

Program output:

```
[(4, 2), (6, 2), (1, 4), (2, 5), (3, 5), (5, 7)]
```

As you can see based on the example above, the sorting is stable. This means that two elements with the same key will be sorted according to the original order. This is a very useful feature when you want to sort a dictionary by several columns.

The value of the key parameter should be a function that takes a single argument and returns a key to use for sorting purposes. Other possible examples of defining the key parameter are given below.

```
#Sorts the string converted to lowercase
print(sorted("Priscilla ra is the best tool for distance
learning".split(), key=str.lower))
```

Program output:

```
['best', 'distance', 'for', 'is', 'learning', 'Priscilla',
'ra', 'the', 'tool']
```

```
#Names sorted by second letter
names = ["David", "Philip", "Jan", "Tom", "Mark"]
bySecondLetter = sorted(names, key=lambda x: x[1])
print(bySecondLetter)
```

Program output:

```
['David', 'Jan', 'Mark', 'Philip', 'Tom']
```

```
#Dictionary sorted first by place of occurrence, then by name
Tuple=[(1,'antelope','land'),
      (2,'dolphin','water'),
      (3,'elephant','land'),
      (4,'catfish','water'),
      (5,'carb','water')]
print(sorted(Tuple, key=lambda x: (x[2],x[1])))
```

Program output:

```
[(1, 'antelope', 'land'), (3, 'elephant', 'land'), (5, 'carb',
'water'), (4, 'catfish', 'water'), (2, 'dolphin', 'water')]
```

If we want to sort in descending order, we use the reverse parameter with a boolean value. This applies to both the function sorted() and the sort method.

```
print(sorted([4,5,5,2,7,2], reverse=True))
```

Program output:

```
[7, 5, 5, 4, 2, 2]
```

```
List=[4,5,5,2,7,2]
List.sort(reverse=True)
print(List)
```

Program output:

```
[7, 5, 5, 4, 2, 2]
```

2.2.5

Please check all true sentences.

- The dictionary can be sorted by only one variable
- Sorting with the function sorted() and the method sort is stable.
- The function sorted() changes the order of the elements in the original list.
- The key parameter should be determined by some function, for example, a lambda function.

2.2.6

Check the result of the following code.

```
library=[(2020,'Clean Code in Python - Second Edition: Develop maintainable and efficient code','Mariano Anaya',1,422),
       (2022,'Python 3. The Comprehensive Guide','Johannes Ernesti, Peter Kaiser',2,1036),
       (2015,'Some methods for Python','Hilip Nowak',2,312),
       (2020,'Mathematics for Machine Learning',' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong',3,398),
       (2015,'The Foundations of Mathematics','Ian Stewart, David Tall',2,416)]

changed=sorted(library, key=lambda x: (x[3],x[0],x[2][1]))
print(changed)
```

Program output:

```
[(2020, 'Clean Code in Python - Second Edition: Develop maintainable and efficient code', 'Mariano Anaya', 1, 422), (2015, 'Some methods for Python', 'Hilip Nowak', 2, 312), (2015, 'The Foundations of Mathematics', 'Ian Stewart, David Tall', 2, 416), (2022, 'Python 3. The Comprehensive Guide', 'Johannes Ernesti, Peter Kaiser', 2, 1036), (2020, 'Mathematics for Machine Learning', ' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong', 3, 398)]
```

- `[(2020, 'Clean Code in Python - Second Edition: Develop maintainable and efficient code', 'Mariano Anaya', 1, 422),`
- `(2015, 'The Foundations of Mathematics', 'Ian Stewart, David Tall', 2, 416),`
- `(2015, 'Some methods for Python', 'Hilip Nowak', 2, 312),`
- `(2022, 'Python 3. The Comprehensive Guide', 'Johannes Ernesti, Peter Kaiser', 2, 1036),`
- `(2020, 'Mathematics for Machine Learning', ' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong', 3, 398)]`
-
- [(2020, 'Clean Code in Python - Second Edition: Develop maintainable and efficient code', 'Mariano Anaya', 1, 422),

  (2015, 'Some methods for Python', 'Hilip Nowak', 2, 312),

(2015, 'The Foundations of Mathematics', 'Ian Stewart, David Tall', 2, 416),

(2022, 'Python 3. The Comprehensive Guide', 'Johannes Ernesti, Peter Kaiser', 2, 1036),

(2020, 'Mathematics for Machine Learning', ' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong', 3, 398)]

- `[(2020, 'Clean Code in Python - Second Edition: Develop maintainable and efficient code', 'Mariano Anaya', 1, 422),`
- `(2020, 'Mathematics for Machine Learning', ' Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong', 3, 398),`
- `(2015, 'Some methods for Python', 'Hilip Nowak', 2, 312),`
- `(2015, 'The Foundations of Mathematics', 'Ian Stewart, David Tall', 2, 416),`
- `(2022, 'Python 3. The Comprehensive Guide', 'Johannes Ernesti, Peter Kaiser', 2, 1036)]`

Regular Expressions

Chapter 3

3.1 Regular expression

3.1.1

Regular expression

Regular expressions are a universal way to find or match (often complex) character strings in text. Single expressions are specified as regex expressions.

In its simpler form, it's a simple pattern that describes a specific searchable text.

The creator of the concept of regular expressions is considered to be the American mathematician Stephen Cole Kleen, who formalized them in the 1950s. Regular expressions were used in two ways to search for patterns in text, and for lexical analysis in compilers.

Technically, a regular expression is a set of rules that a string should follow. If the string satisfies the given rules, we say it matches a regular expression (called to match). Mathematically, a regular expression defines a (usually infinite) set of strings (the elements of the set match the expression).

3.1.2

Basic syntax

Regular expression elements:

text matching

- Each character except special characters is self-defining,

e.g.:

"a" specifies a string consisting of the character a

- Consecutive symbols mean that exactly these must occur in the string symbols in exactly the same order,

e.g.:

"ab3" stands for a string consisting of the letters a, b and the number 3

"regular expression" means a regular expression

When using regular expressions, standard syntax (special symbols) is used, which can be divided into three groups: characters, quantifiers and groupings.

Characters

Characters are simply a way of recording what character we are looking for in the text.

- \d – any number
- \D – not a number
- \w – any letter
- \W - not a letter
- \s – white space
- \S – no white space
- . – any character
- \ – output character
- \b – word boundary
- \B – not a word boundary
- ^ – beginning of the string
- $ – end of string

```
#Example 1 - find letter 'o' in text
import re
text = 'John has a dog, and Ann has a cat and a parrot'

print(re.findall('o', text))
```

Program output:

```
['o', 'o', 'o']
```

```
# Example 2 - we would like to split text which contains a different number of whitespaces (spaces, tabs, newlines)
import re
text = 'one two   three\nfour\tfive'

div = re.split('\s+',text)
print(div)
```

Program output:

```
['one', 'two', 'three', 'four', 'five']
```

3.1.3

Quantifiers

* – 0 or more

+ – 1 or more

? – 0 or 1

{} – exact number of characters

{min, max} – the range of the number of characters

- "ca?t" - matches the patterns ct, cat
- "ca*t" - matches the patterns ct, cat, caat, caaat, ...
- "ca+t" - matches the patterns cat, caat, caaat, ...

```
# Example 3
import re
words = 'aa ab abb abbbb ac abcd abbc accd acd acdb acabb'
# find all words, wherein "a" is followed by "b" zero, one or
more times
print(re.findall('ab*', words))
# find all words, wherein "a" is followed by "b", and "b"
must appear at least once
print(re.findall('ab+', words))
```

Program output:

```
['a', 'a', 'ab', 'abb', 'abbbb', 'a', 'ab', 'abb', 'a', 'a',
'a', 'a', 'abb']
['ab', 'abb', 'abbbb', 'ab', 'abb', 'abb']
```

3.1.4

Grouping

[ ] – matches all characters in brackets

[^] - matches all characters outside the brackets

() – grouping

| - or

```
# Example 4 - look for a string starting with "a" followed by
any character followed by a number.
import re
text = 'John has a 2 dogs, and Ann has a cat and a 9 parrots'
print(re.findall('a.[0-9]', text))
```

Program output:

```
['a 2', 'a 9']
```

3.1.5

"^AI" - will find us a string

- starting with "AI"
- finishing with "AI"

- containing AI

  without AI

3.1.6

which word will be searched with the regular expression „a?b+$"

- acabb
- abbc
- abcd
- aaaaa

3.1.7

Regular expressions in Python, the re library

Python has a built-in package called **re** that you can use to work with regular expressions.

Import the **re** module:

import re

The **re** module offers a set of functions that allow us to search for the appropriate string:

findall - returns a list containing all matches,

search - Returns a Match object if there is a match anywhere in the string

split - returns a list where the string has been split on each match,

sub - replaces one or more matches with a new string.

```
# Example 1
import re

text = "i have 123 cats"
# let's try to extract the number 123 from the text
reg = r'\d\d\d'
match = re.search(reg, text)
# We're checking to see if we found anything
if match:
    num = match.group()
    print(num)
```

Program output:

```
123
```

```
#Example 2 - regular expression responsible for IP search:
import pandas as pd
import re

df = pd.DataFrame(['0.0.0.0',
                   '54.239.128.212',
                   '256.256.256.256',
                   '1.a.255.255',
                   'IP adress hide in text (89.78.209.42).',
                   ],
             columns=['ip'])

ip_pattern = "(?:(?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.){3}\
(?:25[0-5]|2[0-4]\d|[01]?\d\d?)"

df['ip_found'] = df['ip'].apply(lambda x:
re.findall(ip_pattern, str(x)))

print(df)
```

Program output:

```
                                      ip         ip_found
0                                0.0.0.0        [0.0.0.0]
1                         54.239.128.212 [54.239.128.212]
2                        256.256.256.256               []
3                            1.a.255.255               []
4   IP adress hide in text (89.78.209.42).   [89.78.209.42]
```

3.1.8

to return all matches you will use the command:

- findall
- find
- split
- search
- searchall

3.2 Special sequences, character sets

Special sequence means predefined classes of characters that have a unique meaning. This is a useful way to match and search strings for specific conditions. Below we introduce special sequences and their meanings, as well as examples of their use.

Special sequence: \A - Matches pattern only at the start of the string

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\APri", str))
if re.findall("\APri", str):
    print("The sentence begins with 'Pri'")
else:
    print("The sentence does not begin with 'Pri'")
```

Special sequence: \Z Matches pattern only at the end of the string

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("ing\Z", str))
if re.findall("ing\Z", str):
    print("The sentence ends with 'ing'")
else:
    print("The sentence does not end with 'ing'")
```

Special sequence: \b Matches pattern at the beginning or at the end of a WORD. It depends on the use. We have to use it with a raw string prefix 'r'.

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall(r"ool\b", str))
if re.findall(r"ool\b", str):
    print("There is at least one word ending with 'ool'")
else:
    print("There isn't any word ending with 'ool'")
```

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall(r"\bt", str))
if re.findall(r"\bt", str):
    print("There is at least one word starting with 't'")
```

```
else:
    print("There isn't any word starting with 't'")
```

Special sequence: \B Matches pattern if it is not at the beginning or at the end of a WORD. It depends on the use. We have to use it with a raw string prefix 'r'.

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall(r"\Bool", str))
if re.findall(r"\Bool", str):
    print("There is at least one word that has 'ool' in it,
but not at the beginning")
else:
    print("There isn't any word that has 'ool' in it, but not
at the beginning")
```

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall(r"ool\B", str))
if re.findall(r"ool\B", str):
    print("There is at least one word that has 'ool' in it,
but not at the end")
else:
    print("There isn't any word that has 'ool' in it, but not
at the end")
```

Special sequence: \d - Matches to any digit: 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\d", str))
if re.findall("\d", str):
    print("There are digits")
else:
    print("There are no digits")
```

Special sequence: \D - Matches to any non-digit

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\D", str))
if re.findall("\D", str):
    print("There are no digits")
else:
    print("There are only digits")
```

Special sequence: \s - Matches any whitespace character

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\s", str))
if re.findall("\s", str):
    print("There is at least one whitespace")
else:
    print("There is no whitespace")
```

Special sequence: \S - Matches any non-whitespace character

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\S", str))
if re.findall("\S", str):
    print("There is at least one charakter that is not a
whitespace")
else:
    print("There are only whitespaces")
```

Special sequence: **\w - Matches any alphanumeric character: 0,...,9, a,...,z,A,...,Z**

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\w", str))
if re.findall("\w", str):
    print("There is at least one alphanumeric charakter")
else:
    print("There is no alphanumeric charakter")
```

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\w{7}", str))
if re.findall("\w{7}", str):
    print("There is at least one word with at least 7
alphanumeric characters")
else:
    print("There is no word with at least 7 alphanumeric
characters")
```

Special sequence: \W - Matches any non-alphanumeric character: 0,...,9, a,...,z,A,...,Z

```
import re
str = "Priscilla is the best tool for distance learning"
print(re.findall("\W", str))
```

```
if re.findall("\W", str):
    print("There is at least one non-alphanumeric charakter")
else:
    print("There is no non-alphanumeric charakter")
```

3.2.2

Check the special sequence, which will allow us to check if there is any whitespace in the string.

```
• str = "I love Priscilla!"
• print(re.findall("\s", str))
• str = "I love Priscilla!"
• print(re.findall("\S", str))
• str = "I love Priscilla!"
• print(re.findall("\w", str))
• str = "I love Priscilla!"
• print(re.findall("\W", str))
• str = "I love Priscilla!"
• print(re.findall("\b", str))
• str = "I love Priscilla!"
• print(re.findall("\b", str))
• str = "I love Priscilla!"
• print(re.findall("\d", str))
• str = "I love Priscilla!"
• print(re.findall("\D", str))
```

3.2.3

Check the special sequence, which will allow us to check if there is a word that has 'l' but not at the beginning of the word (in the middle or at the end).

```
• str = "I love Priscilla!"
• print(re.findall(r"\Bl", str))
• str = "I love Priscilla!"
• print(re.findall(r"l\B", str))
• str = "I love Priscilla!"
• print(re.findall("\Bl", str))
• str = "I love Priscilla!"
• print(re.findall("\Al", str))
• str = "I love Priscilla!"
• print(re.findall("l\Z", str))
• str = "I love Priscilla!"
• print(re.findall(r"\Al", str))
• str = "I love Priscilla!"
• print(re.findall(r"l\Z", str))
• str = "I love Priscilla!"
• print(re.findall("\D", str))
```

3.2.4

Check the output of the code below

```
import re
str = "I love Priscilla! She inspires me..."
print(len(re.findall("\W", str)))
```

- 0
- Yes
- 27
- 9
- 4
- 6

3.2.5

Character classes can be used instead of the special sequence. In many cases, they can be used equivalently. The basic form of defining character classes is to write a string of characters in square brackets.

For example, the character class [tc]ool, it's a match for words tool and cool.

```
import re
str = "Priscilla is the best tool for distance learning. It is
so cool."
print(re.findall("[tc]ool", str))
```

We can build the character classes by using a range of values, we should use '-' then.

For example, [d-h] means all letters from d through h. Or [4-7] means the digits 4,5,6,7.

Several ranges can be used for one character class definition. For example, [d-hk-m] means all letters from d through h and k through m.

We can also build the character classes by using a negation, we use the sign '^' for negation.

For example, [^d-h] means all letters except d through h. Or [^4-7] means the digits 0,1,2,3,8,9.

```
import re
str = "Priscilla is the best tool for distance learning. She
is so cool."
#Match d, e, f, g, h, a, or c.
```

```
print(re.findall("[d-ha-c]", str))
#Match a letter other than a, b, c and d.
print(re.findall("[^abcd]", str))
#Match d, e, f, g or h followed by a, b or c.
print(re.findall("[d-h][a-c]", str))
#Match d, e, f, g or h followed by a letter other than a and
b.
print(re.findall("[d-h][^ab]", str))
```

Thus, many special sequences can be used equivalently with character classes.

The equivalent pairs are listed below.

- \d - character class [0-9]
- \D - character class [^0-9]
- \s - character class [\t\n\x0b\r\f]
- \S - character class [^ \t\n\x0b\r\f]
- \w - character class [a-zA-Z_0-9]
- \W - character class [^a-zA-Z_0-9]

```
import re
str = "Priscilla is the best tool for distance learning. She
is so cool."
print(re.findall("\W", str))
print(re.findall("[^a-zA-Z_0-9]", str))
```

3.2.6

Check all true sentences.

- The following two codes have the same meaning.

```
import re
str = "Artificial intelligence will change our world!"
print(re.findall("\d", str))
import re
str = "Artificial intelligence will change our world!"
print(re.findall("[^0-9]", str))
```

- The output of the following code will be 40

```
import re
str = "Artificial intelligence will change our world!"
print(len(re.findall("[^1-9! ]", str)))
```

- The output of the following code will be ['Ar']

```
import re
str = "Artificial intelligence will change our world!"
print(re.findall("[a-iA-I][^a-n !]", str))
```

3.2.7

match function - the much function attempts to match a pattern, defined as regular expression, to text (re.match(pattern, string, flags)). You may specify more than one flag, by separating them with the pipeline sign (|). The re.match function returns a match object on success and None upon failure.

Flags:

re.I - Performs case-insensitive matching

re.L - Interprets words according to the current locale

re.M - Makes $ match the end of a line and makes ^ match the start of any line

re.S - Makes a period (dot) match any character, including a newline

re.U - Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B

re.X - It ignores whitespace (except inside a set [] or when escaped by a backslash and treats unescaped # as a comment marker

```
#Finds Priscilla-Fan, Priscillafan, PRISCILLA-FAN, etc.

import re
pattern = r'Priscilla[- ]?fan.'

texts=[ 'Priscilla-Fans live everywhere', 'Priscillafan lives
here', 'Samantafan lives here' ]
for text in texts:

  print(text)

  if re.match(pattern, text, re.IGNORECASE): print('Match')
  else: print('No match')

  print()
```

Program output:

```
Priscilla-Fans live everywhere
Match
Priscillafan lives here
Match
Samantafan lives here
No match
```

Groups - match function may also be used to capture groups

```
#Match dates formatted like MM/DD/YYYY, MM-DD-YY, etc.

import re
pattern = re.compile(r'^(\d\d)[-/](\d\d)[-/](\d\d(?:\d\d)?)$')

dates = [ '02/14/2023', '05-01-1993', '30th July 2022']
for date in dates:

  print(date)
  obj = pattern.match(date)

  if obj:

    print('Match')

    month = obj.group(1)  #02
    day = obj.group(2)  #14
    year = obj.group(3)  #2023

    print(day)
    print(month)
    print(year)
    print()

  else: print('No match')
```

Program output:

```
02/14/2023
Match
14
02
2023

05-01-1993
Match
01
05
1993

30th July 2022
No match
```

3.2.8

Write Python expression which verifies whether the given time matches one out of a few specified formats and extract the value of hours and minutes.

Random Number Generation

Chapter 4

4.1 Library random - generating random numbers

4.1.1

Random numbers

Often when writing machine code comes the use of data generation. Random number generators can be used for this purpose.

In Python, the most popular library for generating values is the random library

let's take a look at the functions offered by the random library:

```
# Generating a floating random number between 0 and 1
# function random()

import random
nu =random.random()
print(nu)
```

Program output:

```
0.7809914352329894
```

```
# Generating a random integer within a given range – randint()
# randint(min,max) -where [min,max] defin the lower and upper
limit of the range.

import random
nu = random.randint(20,40)
print(nu)
```

Program output:

```
34
```

```
# Generating a random integer list using for loop – randint()
import random
rand_list = []
for i in range(0,5):
  nu = random.randint(0,10)
  rand_list.append(nu)
print(rand_list)
```

Program output:

```
[3, 0, 5, 9, 9]
```

```
# Generating a sample of random integers within a given range
- sample()
# sample(range, how many values)

import random
nu = random.sample(range(0, 50), 10)
print(nu)
```

Program output:

```
[38, 19, 33, 21, 0, 48, 7, 17, 13, 28]
```

```
#Selecting a random number from a given list - choice()

import random
lists = [2, 10, 11, 8, 13, 26, 17]
nu = random.choice(lists)
print(nu)
```

Program output:

```
26
```

```
# Generating a random number from a list in the specified
range - randrange()

import random
nu = random.randrange(10,20,5)
print(nu)
```

Program output:

```
10
```

```
# Generating a floating random number within a given range -
uniform()
#uniform(min, max)

import random
nu - random.uniform(10,15)
print(nu)
```

Program output:

```
10
```

```
# Generating a randomly shuffled list - shuffle()
# a shuffle() function takes a list as an argument and shuffles
the elements

import random
lists=[1, 2, 3, 4, 5, 6, 7, 8]
random.shuffle(lists)
```

```
print(lists)
```

Program output:

```
[3, 8, 2, 4, 7, 1, 6, 5]
```

4.1.2

To change the order of items in a list you can use a function

4.1.3

To select a float from a given range, use the function

Pandas

Chapter 5

5.1 Pandas introduction

5.1.1

The Pandas library is one of the most widely used Python libraries in the world. Pandas stand for "Python Data Analysis Library". Pandas is a freely distributable library.

The significant advantage of Pandas is that it can take data (e.g. in CSV format) and create a table structure object called a *DataFrame*. This structure is very similar to what we can do in any spreadsheet editor (e.g. Excel). Working with a spreadsheet is much easier than working with dictionaries or lists, which require using loops to search through them.

Nowadays, most tools for working with Python already include the Pandas library directly installed in the base version, so installing the library is unnecessary and we will not describe it.

5.1.2

However, in order to use the Pandas library in our program, we need to **import** it. Importing the library means loading it into memory, where we can then work with it. A similar procedure applies to any library you want to use in Python (e.g. numpy, matplotlib, etc.).

```
import pandas
```

As a priority, the library import is called using the "**import**" command. But then we have to use the full library name, i.e. *pandas*, in each call. Therefore, there is an option to create an *alias* using the "**as**" command. So in our case, the variable "**pd**" will represent the pandas library.

```
import pandas as pd
```

5.1.3

Which of the import calls for pandas library are correct?

- `import pandas`
- `import pandas as pd`
- `import pandas as panda`
- `as panda import pandas`
- `import panda`
- `import pd as pandas`
- `import pd`

5.1.4

One of the main reasons for the popularity of pandas is its ability to handle different types of data:

- spreadsheets with columns that are capable of storing different types of data (e.g., numeric data and text data)
- ordered and unordered series data (any sequence of numbers in a list, e.g. [2,4,8,9,10])
- multi-dimensional matrix data (three-dimensional, four-dimensional, etc.)
- any other form of statistical data (e.g., data from databases)

In addition, the large number of intuitive and easy-to-use functions/methods makes pandas a convenient and frequently chosen tool for data analysis.

5.1.5

Before we start working with the pandas library in more detail, let's recall some of the basic structures we'll be working with.

List (Series)

In pandas, the Series object represents one-dimensional data. It is initialized with the pd.Series() statement. In the following example, you can see how to create a numeric list using pandas Series:

```
import pandas as pd
s = pd.Series([10,11,12,13,15])
print(s)
```

We can see that we have a one-dimensional list as an output that is represented by pandas Series. The numbers on the left, as if in the first column, represent the index of each element of the list.

5.1.6

Create a Pandas Series from the following list of numbers:

```
[15,0,2,19,200,10,22]
```

What will be the average of the given values? Do not round the result.

Procedure to solve the problem:

- import the pandas library
- create a variable of type pandas Series
- use a for loop to go through the Series and add the numbers

- divide the sum of the numbers by their count and print the average

5.2 Data input

5.2.1

As mentioned earlier, the Pandas library can store data in the form of a table called a **DataFrame**. A **DataFrame** is a two-dimensional representation of data in rows and columns that can be initialized in pandas using the **DataFrame()** function. In the following code, a simple list is converted to a one-dimensional DataFrame:

```
import pandas as pd

df = pd.DataFrame([10,11,12,13,15])
print(df)
```

As we can see, the result is similar to the Series view but remember that in the case of Series it is only a one-dimensional list. On the other hand, a DataFrame is similar to a table, so in our example, we again see first the indexes in the first column, which denote the rows, and in the first row, we see the index or the column name. This is what makes the difference from Series, where we didn't have column naming. The way the pandas library works is that unless we specify column information when we create the DataFrame, it automatically numbers the columns starting from 0.

5.2.2

Working with DataFrame provides various options. Among the first functions we will introduce is the **shape()** function.

```
import pandas as pd

df = pd.DataFrame([11,12,13,45,202,71,239,3,5])

print(df.shape)
```

The **shape** function returns information about the shape of the created DataFrame. So our DataFrame consists of 9 rows and 1 column.

5.2.3

We mentioned that by default the columns are named with numeric indices. By calling the **columns** method, we can either write our own column naming or print the column names of our DataFrame.

```
import pandas as pd

df = pd.DataFrame([11,2,8,99,3,74,123,3])

print(df.columns)
df.columns = ['num']

print(df.columns)
print(df.index)
```

Printing the **columns**, in the case of the default option, will output information that it is a range of indexes that start with 0 and end with 1, with a step of 1. We get similar information for the row indexes when we call the **index** method. However, the moment we assign a new list to the columns, in our case with the name of a single column, the **columns** method will print the index list with our name on it. Analogously, we can also change the names of rows in a DataFrame.

```
import pandas as pd

df = pd.DataFrame([11,2,8,99,3,74,123,3])
df.columns = ['num']
df.index =
['row1','row2','row3','row4','row5','row6','row7','row8']
print(df)
```

5.2.4

Until now we have been working with a one-dimensional list but DataFrame is more often used as a two-dimensional list or table. So let's be given two integer lists:

```
[36,22,3,78,2,1]
[4,66,81,22,3,7]
```

Store both lists in a DataFrame and see what its shape is. Creating a DataFrame is analogous to creating a one-dimensional list but now we'll put it there as a list of lists **[[...],[...]]**.

```
# import library

# create dataframe

# examine
```

- (6,2)
- (2,6)
- [6,2]

- [2,6]

5.2.5

Working with pandas involves importing data from different data files and writing back the output in different formats. These operations are essential processes when working with data, and the most common type of file encountered is the CSV file. Reading a CSV file and then transforming it into a DataFrame is built into the pandas library using the **read_csv()** function. Suppose we have a *students.csv* file stored on disk, then loading it into a DataFrame will look like this:

```
import pandas as pd

df = pd.read_csv('students.csv', sep=';')
```

The second parameter in the function is the so-called separator (**sep**). A standard CSV file has individual records separated by semicolons. In the case of a specific file, we can of course modify the separator to the desired character or set of characters. The pandas library then loads the contents of the file into the DataFrame based on the specified delimiter and it is the delimiter that is used to create the columns.

Pandas also supports other file types such as XLS, JSON, XML, or HDF5.

5.2.6

We don't have to load the CSV file only from our own disk but we can also load it from the web storage. We just need to pass the direct URL to the requested CSV file as the first parameter to the **read_csv()** function. So, load the titanic.csv data file from the following URL into the DataFrame and output its shape. The separator in this case is a **comma**.

```
https://priscilla.fitped.eu/data/pandas/titanic.csv

# import library

# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
```

5.2.7

There are many cases where we need to write data to a file and save it to disk for future use. The pandas library uses the **to_csv()** function to write data to a file. The parameter of the called function is the path to the file where the data is to be written. If the file already exists on the disk it overwrites it with the new content. Similar to loading data, we can use the **sep** parameter to specify a separator to separate our data. It is also convenient to specify an additional parameter, **index**,

which has two options, *True* or *False*. Index tells whether to include the row index in the result file, if we don't specify this value the default is *True*.

```
import pandas as pd

df = pd.DataFrame([[22,1,34],[11,2,3]])
df.to_csv('export.csv', sep = ';', index = False)
```

5.2.8

Pandas also allows you to import data directly from an MS Excel file. The condition for working with such a file is that it does not contain any images, formulas or macros. For this reason, we also more often encounter that the MS Excel file is saved in CSV format. However, if we are going to work with an XLS/XLXS file, we proceed as follows:

```
import pandas as pd

df = pd.read_excel('file.xlsx', sheetname='Sheet1')
```

The *sheetname* argument defines the name of the sheet from which we will retrieve data.

5.2.9

Nowadays, JSON format is also very popular, so we have a library and a set of methods for working with this flexible format. JSON is a lightweight data interchange format that was inspired by JavaScript. Importing JSON data using the *json* library not only allows you to specify the path to the file in the operating system or cloud storage but also via a URL. In this way, for example, we can connect to increasingly popular sites that provide datasets from various domains. The following command displays logs from using the repository.

```
import pandas as pd
import json

df =
pd.read_json('https://api.github.com/repos/pydata/pandas/issue
s?per_page=5')

print(df.head())
```

5.3 Checking the data

5.3.1

Working with a DataFrame starts with loading the dataset and before we look at the different options for working with the data, it's a good idea to look at what data we've actually stored in the DataFrame. The first step is the already introduced **shape** function, which tells us how many rows and columns the DataFrame has created.

If we have a dataset that contains thousands of rows, it's not a good idea to list all the rows. In such cases, we can get an overview of the dataset by listing only the first few rows. This is what the **head()** and **tail()** functions are for.

In the theoretical examples, we'll work with the file *scrabble_games.csv*, which keeps track of tournament games in the board game Scrabble, keeping track of the winners, their scores, and other interesting information about each game.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', delimiter=',')
print('Function head:')
print(df.head())
print('Function tail:')
print(df.tail())
```

If we want to display the first few rows of data, we can use the **df.head()** command, where *df* is the name of the DataFrame. Similarly, if we want to display the last few rows, we can use the **df.tail()** command. In both of these cases, the first (or last) **5 rows** are displayed by default.

If we want to display more (or fewer) rows, we can specify the number of rows as a parameter in the **head()** or **tail()** functions. For example, the first 10 rows can be displayed using the **df.head(10)** function. To display the last eight rows we use the **df.tail(8)** function.

5.3.2

Load the dataset regarding titanic. Practical tasks will mostly work with this dataset, which contains the passenger records of the infamous steamship Titanic.

Use the **head()** and **tail()** functions to examine the data and identify the correct age of the passengers (Age column):

- 7th passenger from the beginning
- 12th passenger from the end

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

- 54
- 56
- 25
- 22
- 28
- 35
- 2

5.3.3

Pandas supports various data types such as **int64**, **float64**, **date**, **time**, and **Boolean** values. When analyzing data, there will be countless times when we need to convert data from one type to another. Similarly, the moment we load a dataset into a DataFrame, it's a good idea to familiarize yourself with what data type is actually in each column. In such cases, it is essential to understand these different data types.

The first aspect to be aware of when working with any data is the different data types. We can get these using the **dtypes** method.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep = ',')
print(df.dtypes)
```

From the output, we can see that our dataset contains three different data types: **int64**, **bool** and **object**. The object specifies textual data or a combination of textual and numeric data. int64 represents integer data and bool in turn speaks of a logical true/false value.

5.3.4

Load the data from the dataset titanic.csv. Examine the data types in the dataset and select which data type predominates in the dataset.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

- int64
- object

- float64

5.3.5

Similar information as in the previous assignment we can also get using the **info** function. In this case, however, we get much more information, not only about the individual columns (variables) but also about the entire DataFrame.

```
import pandas as pd
df = 
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df.info())
```

After calling the **info** function, we see that our dataset consists mostly of integer data. We can also see that the dataset contains 19 columns and 99402 non-zero rows.

Data Manipulation

Chapter 6

6.1 Missing values

6.1.1

When working with real datasets, you will certainly often encounter missing data during data analysis. Understanding how the pandas program displays missing data for each data type is very important to ensure the accuracy of our future data analysis. Missing values can be the most unwanted values in data science. Since it is not reasonable to ignore missing values, we need to find ways to work with them efficiently and correctly. Pandas is quite flexible in handling missing values. Empty values are often denoted by **NaN**.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
print(df.info())
```

At first glance, we may not see anything extraordinary in the displayed data but if we look closely at the third column in the displayed table, not every row contains all 24 non-zero records. That means all columns except the date contain several non-zero or missing values. We can try to verify this by displaying the DataFrame header:

```
print(df.head())
```

6.1.2

Load the data from the dataset titanic.csv. Examine the data in the dataset and select which columns **contain** the missing values.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

- Age
- Cabin
- PassengerId
- Survived
- Name
- Ticket

6.1.3

The first option to deal with missing values is to remove either rows or columns that contain any missing values. If we want to remove rows that contain empty values, we call the **dropna()** function.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.dropna()
print(df)
```

In this way, we removed all rows that contained at least one missing value. The second option is to remove the columns that contain missing values by setting the **axis** parameter to 1. Axis specifies the axis along which the function is to be executed.

```
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.dropna(axis=1)
print(df)
```

The disadvantage of both approaches is that we also delete the records that have a value from the data matrix. When we delete the columns, we end up with only the date variable, which in principle does not tell us anything of substance but satisfies the condition that it is the only one that does not have missing values. There is also another option and that is to delete only the rows or columns that contain only missing values. We can do this using the **how** parameter. In our example, since the Date column does not contain any missing values, calling this function will not delete any rows or columns.

```
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.dropna(how="all")
print(df)
```

6.1.4

Load the data from the dataset titanic.csv. Examine the data in the dataset and remove rows that **contain** missing values. How many rows are left in the cleaned dataset?

```
# import library
```

```
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# drop rows with missing values
```

6.1.5

Deleting all or some records is not always the best way to go. Sometimes we want to focus more on deleting rows or columns with a few or mostly missing values. We can't verbalize this to pandas in the form of a how parameter but we can use the **thresh** parameter, which specifies the value of the number of missing records in a row/column that we keep in the matrix. Other rows/columns with missing values that do not meet this condition will be removed from the matrix.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.dropna(thresh=4)
print(df)
```

Since we have a data matrix with 6 columns, if we set the threshold to 4 it will mean that rows that contain 3 or more missing values will be deleted. This way we have deleted only one row from the original DataFrame.

6.1.6

Until now, we have not looked at which column to delete missing values from but we can also specify individual columns. Using the **subset** parameter, we can restrict the deletion of missing values to the columns (variables) we specify.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.dropna(subset=['Waste','Gas'])
print(df)
```

Program output:

```
          Date  Household  Electricty  Water  Gas  Waste
7   2020-08-31     1748.0         NaN    7.8  8.9    8.0
8   2020-09-30     1748.0        18.4    2.4  4.5    3.0
9   2020-10-31        NaN         NaN    9.6  6.0    4.0
10  2020-11-30     1748.0         5.8   16.6  1.7    7.0
11  2020-12-31     1748.0        14.6    8.4  2.1    3.0
12  2021-01-31     1748.0        14.8   17.6  9.8    3.0
13  2021-02-28     1748.0        10.8    7.0  0.5    3.0
14  2021-03-31     1748.0        13.4    2.4  8.2    6.0
```

```
15  2021-04-30     1748.0         13.8     2.4  3.6    7.0
16  2021-05-31     1748.0          1.4     9.2  4.9    2.0
17  2021-06-30     1748.0         19.0    11.4  9.0    6.0
18  2021-07-31     1748.0          8.4     8.0  2.6    5.0
19  2021-08-31     1748.0          8.6     0.8  4.4    1.0
20  2021-09-30     1748.0         14.8    13.4  0.4    2.0
21  2021-10-31     1748.0          0.8     9.2  4.5    5.0
22  2021-11-30     1748.0         10.8    13.2  1.7    8.0
23  2021-12-31     1748.0          2.2     4.8  3.5    7.0
```

6.1.7

We have introduced various methods of deleting rows or columns based on missing values. Deleting is not the only option. In some cases, instead of deleting missing values, we can choose to add or replace them. In fact, the filling may be a better option because data implies value and, as we have seen, in some cases we have lost most of the original data set. Of course, the method of filling in missing values depends on the data structure and the task. The **fillna** function is used to fill in missing values. The first option is to fill in a constant value. If one constant value is sent to the function, all missing values are replaced by that constant. Therefore, it is preferable to specify different constant values for different columns, which we can send as a parameter in the form of a dictionary.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
val = {"Household": 1748, "Waste": 0}
df = df.fillna(value = val)
print(df)
```

The value for Household has been adjusted to 1748, indicating a given household and the expenditure for Waste has been set to 0, as we cannot estimate that value.

6.1.8

Load the data from the dataset titanic.csv. Examine the data in the dataset and complete the rows in the **Age** column that **contain** missing values. Insert a constant value of 0 in place of the missing values. In the code, you have an output of the average age of the passengers. How much did the result change after the empty values were filled in? (write the answer in integer)

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# print mean with missing values
```

```
print(df['Age'].mean())
# fill Age with 0
# print mean with filled values
print(df['Age'].mean())
```

6.1.9

As you may have noticed in the previous task, adding a certain constant does not change our results. Another option for filling in missing values is to use aggregate functions. This way we can at least partially estimate what the approximate values should have been in place of the missing values. We can use the **mean**, **median** or **mode** function.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.fillna(df['Waste'].mean())
print(df)
```

Program output:

```
          Date    Household  Electricty      Water       Gas
Waste
0   2020-01-31  1748.000000    7.000000  13.000000  5.700000
4.705882
1   2020-02-29  1748.000000   10.600000  18.200000  7.200000
4.705882
2   2020-03-31     4.705882    4.705882   4.705882  4.705882
4.705882
3   2020-04-30  1748.000000   18.600000  18.800000  4.705882
4.705882
4   2020-05-31  1748.000000   12.000000  14.200000  4.800000
4.705882
5   2020-06-30  1748.000000   12.800000   8.400000  8.300000
4.705882
6   2020-07-31  1748.000000   10.800000  16.400000  5.100000
4.705882
7   2020-08-31  1748.000000    4.705882   7.800000  8.900000
8.000000
8   2020-09-30  1748.000000   18.400000   2.400000  4.500000
3.000000
9   2020-10-31     4.705882    4.705882   9.600000  6.000000
4.000000
10  2020-11-30  1748.000000    5.800000  16.600000  1.700000
7.000000
```

```
11   2020-12-31   1748.000000   14.600000    8.400000   2.100000
3.000000
12   2021-01-31   1748.000000   14.800000   17.600000   9.800000
3.000000
13   2021-02-28   1748.000000   10.800000    7.000000   0.500000
3.000000
14   2021-03-31   1748.000000   13.400000    2.400000   8.200000
6.000000
15   2021-04-30   1748.000000   13.800000    2.400000   3.600000
7.000000
16   2021-05-31   1748.000000    1.400000    9.200000   4.900000
2.000000
17   2021-06-30   1748.000000   19.000000   11.400000   9.000000
6.000000
18   2021-07-31   1748.000000    8.400000    8.000000   2.600000
5.000000
19   2021-08-31   1748.000000    8.600000    0.800000   4.400000
1.000000
20   2021-09-30   1748.000000   14.800000   13.400000   0.400000
2.000000
21   2021-10-31   1748.000000    0.800000    9.200000   4.500000
5.000000
22   2021-11-30   1748.000000   10.800000   13.200000   1.700000
8.000000
23   2021-12-31   1748.000000    2.200000    4.800000   3.500000
7.000000
```

In our case, we supplemented the Waste column with the average value measured from the other rows.

6.1.10

We have shown different ways of filling in values, the last way is to fill in based on the previous or next value from the given column. This method can come in handy when working with time series data. Consider that we have data that contains, for example, daily temperature measurements and the temperature on one day is missing. The optimal solution would be to use the temperature on the following or the previous day.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df = df.fillna(method='bfill')
print(df)
```

The **bfill** command fills in missing values backwards, so they are always replaced by the next value. The opposite approach is **ffill**, which fills in values based on the previous available value. However, such a call will fill in all missing values, which may not always be appropriate. Therefore, it is possible to **limit** the number of missing values filled in by this method. If we set the limit parameter to 1, it means that only the next value is always completed.

```
df = df.fillna(method='bfill', limit=1)
```

6.2 Data selection

6.2.1

Pandas provides several methods for selecting data from a DataFrame but we'll start by learning how to index using two functions: **loc** and **iloc**. Pandas allows us to index a DataFrame by rows and columns of integer values (indexed by 0) or namespaces. Despite what we've encountered so far, rows don't always have to be indexed by numbers only and conversely, columns don't always have to be indexed by namespaces. The **iloc** function allows us to use a *numeric* index of rows and columns, on the other hand, **loc** implies the use of *namespaces*. The key idea is that integer values are automatically matched to the number of rows or columns but the namespaces are appended to the rows and columns to which they have been assigned, so if we delete a row or column, it is removed from the sequence.

6.2.2

ILOC

Indexing with **iloc** in the pandas library is used to index or select by position based on integer location. The function parameters are first the row index and then the column index and we can also insert a list of rows or columns when making a selection. Since it works on an integer basis, it represents the selection of rows and columns by number in the order in which they occur in the DataFrame. As mentioned earlier, the default state is that each row has a row number from 0 to the total number of rows -1 and **iloc** operates just based on these indexes. The important thing to remember is that the index always starts with 0.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
# rows:
print(df.iloc[0]) # first row
print(df.iloc[5]) # fifth row
print(df.iloc[-1]) # last row
```

```
# columns:
print(df.iloc[:,0]) # first column
print(df.iloc[:,5]) # fifth column
print(df.iloc[:,-1]) # last column
```

Based on the index, we can dump both the first and the last row from the DataFrame. To do this, we make use of slicing that is used, for example, when working with strings. We work analogously in the case of columns but we have to specify the first parameter indicating the selected set of rows. In this case, we want all rows for a given column, so we specify the **:** character to indicate the selection of all values.

There is one important fact to remember when working with **iloc**. In case we are selecting only one row, we get back a **series**, and in case of selecting more than 1 row, we have a **DataFrame**. If we would like to get a DataFrame in the first case as well, we need to specify a list with one value as a parameter.

6.2.3

Load the data from the dataset titanic.csv. Examine the data in the dataset and select the correct row/column selection boundary to get the first 10 rows with only the passenger name. Use the **iloc** function.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

- `df.iloc[0:10,3]`
- `df.iloc[10,'Name']`
- `df.iloc[0:10,'Name']`
- `df.iloc[10,3]`
- `df.iloc[-10,4]`
- `df.iloc[0:10,4]`
- `df.iloc[10,4]`
- `df.iloc[-10,4]`

6.2.4

Load the data from the dataset titanic.csv. Examine the data in the dataset and select the correct row/column selection bounds to get the following subset of data. In the answer write out the entire command, e.g.

```
df.iloc[:,0:5]
```

	Name	Sex	Age
100	Petranec, Miss. Matilda	female	28.0
101	Petroff, Mr. Pastcho ("Pentcho")	male	NaN
102	White, Mr. Richard Frasar	male	21.0
103	Johansson, Mr. Gustaf Joel	male	33.0
104	Gustafsson, Mr. Anders Vilhelm	male	37.0
...	...	...	...
195	Lurette, Miss. Elise	female	58.0
196	Mernagh, Mr. Robert	male	NaN
197	Olsen, Mr. Karl Siegwart Andreas	male	42.0
198	Madigan, Miss. Margaret "Maggie"	female	NaN
199	Yrois, Miss. Henriette ("Mrs Harbeck")	female	24.0

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

6.2.5

LOC

Unlike iloc, **loc** works on the basis of name labels, or text labelling of row and column indexes. We encounter this more often because especially when selecting columns it is much clearer to call a column based on a given index namespace. However, it is possible to do similarly for rows. Consider our Scrabble game registration dataset. We have named the columns with name labels but the row index is an integer label starting at 0. In the data matrix, we also have information regarding the identifier of an individual game. Let's say that our row indices will just represent that identifier.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
```

```
df2 =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df.head())
df2.set_index('gameid', inplace=True)
print(df2.head())
```

Although visually they are still numbers, calling the **loc** function will mean that the program treats them as text data. While it can handle integer values, calling

```
df.loc[26]
```

it is not the same as calling the iloc function with the parameter 26. In our case, it will return a row whose index is 26, in other words, it will return a row with the game number 26. This can be anywhere in the data matrix; in contrast, the iloc function will always return the 26th row of the matrix.

```
print('LOC:')
print(df2.loc[26])
print('ILOC:')
print(df.iloc[26])
```

6.2.6

However, most often in practice, we encounter data selection based on conditional selection. The so-called Boolean indexing or logical selection is used for this purpose. You enter a list or series of *True/False* values into the **loc** function, based on which rows are selected in which your series has *True* values.

For example, if we take the name of the winner using the condition below, we get a pandas series with *True/False* values for each row, with *True* values for rows where the winner's name is "Harriere Lakernick".

```
df['winnername'] == 'Harriette Lakernick'
```

We can insert these boolean lists directly into the **loc** function.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df.loc[df['winnername'] == 'Harriette Lakernick'])
```

Program output:

```
       gameid  tourneyid    tie  winnerid           winnername
winnerscore  \
```

```
0           1          1  False        268  Harriette Lakernick
0
1           2          1  False        268  Harriette Lakernick
0
2           3          1  False        268  Harriette Lakernick
0
3           4          1  False        268  Harriette Lakernick
0
4           5          1  False        268  Harriette Lakernick
0
...       ...        ...    ...        ...                  ...
...
94679   95278        327  False        268  Harriette Lakernick
381
98413   99012        342  False        268  Harriette Lakernick
0
98414   99013        342  False        268  Harriette Lakernick
0
98415   99014        342  False        268  Harriette Lakernick
0
98416   99015        342  False        268  Harriette Lakernick
0

       winneroldrating  winnernewrating  winnerpos  loserid
losername  \
0                 1568             1684          1      429
Patricia Barrett
1                 1568             1684          1      435
Chris Cree
2                 1568             1684          1      441
Caesar Jaramillo
3                 1568             1684          1      456
Mike Chitwood
4                 1568             1684          1     1334
Nancy Scott
...                ...              ...        ...      ...
...
94679             1799             1718         16      507
James Frankki
98413             1718             1691         14      733
Steve Pellinen
98414             1718             1691         14      674
Laura Scheimberg
```

```
98415             1718             1691          14       675
Ruth Hamilton
98416             1718             1691          14       507
James Frankki

       loserscore  loseroldrating  losernewrating  loserpos
round  division  \
0               0            1915            1872         3
1         1
1               0            1840            1798         6
2         1
2               0            1622            1606        10
3         1
3               0            1612            1600         9
4         1
4               0            1537            1590         4
6         1
...           ...             ...             ...       ...
...       ...
94679         340            1652            1643        14
9         1
98413           0            1840            1843         4
1         1
98414           0            1617            1581        16
3         1
98415           0            1683            1647        15
10         1
98416           0            1643            1651        13
12         1

             date  lexicon
0      1998-12-06    False
1      1998-12-06    False
2      1998-12-06    False
3      1998-12-06    False
4      1998-12-06    False
...           ...      ...
94679  2000-10-14    False
98413  2000-11-11    False
98414  2000-11-11    False
98415  2000-11-11    False
98416  2000-11-11    False

[85 rows x 19 columns]
```

6.2.7

Load the data from the dataset titanic.csv. Examine the data in the dataset and use the **loc** function to determine how many people had a ticket for booth C123.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

6.2.8

Load the data from the dataset titanic.csv. Examine the data in the dataset and use the **loc** function to find out how many people paid more than $30 for a ticket (Fare column).

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

6.2.9

We've demonstrated working with conditional selection but we'll definitely need to create compound conditions. To create a compound condition we can use the logical operators **and** (**&**) or **or** (|). The logic is similar to the case of only one condition, that is, a series of *True/False* results will be produced. In case we use the **or** logical operator, we print the rows for which at least one condition is satisfied. On the other hand, the logical operator **and** means that we only select rows for which all conditions are satisfied at the same time. It is important to enclose each condition in parentheses because without parentheses an inconsistent result may be generated.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df.loc[(df['tie']==True) & (df['division']==2)])
```

Program output:

```
       gameid  tourneyid   tie  winnerid           winnername
winnerscore  \
77         78          1  True       458           Jan Asuquo
0
78         79          1  True       458           Jan Asuquo
0
```

```
137        138          1  True      1330           Carl Davis
0
231        232          1  True      2223        Karen Slaton
0
1138      1139          3  True      2912          Jim Piazza
0
...        ...        ...   ...       ...                  ...
...
95751    96350        329  True       806      John Robertson
0
95799    96398        329  True       910  Michael Krepakevich
0
96107    96706        330  True      2151    Laurina Ghiglione
0
96801    97400        333  True       112      Barbara Lowrey
0
98698    99297        342  True      3671    Vivian Henderlite
0

       winneroldrating  winnernewrating  winnerpos  loserid
losername  \
77                1254             1255         15     2223
Karen Slaton
78                1254             1255         15      465
Carl Hickerson
137               1434             1465          3      486
Thelma Litton
231               1209             1188         23      455
Carole Miller
1138              1466             1455         39     5890
Anita Shields
...                ...              ...        ...      ...
...
95751             1493             1500          5     1279
Danny Panganiban
95799             1453             1475          7      907
Tim Knowles
96107             1277             1238          4     1475
Margaret West
96801             1422             1443          6     1483
Thomas E Wood
98698             1336             1306         17     3171
Rita McGee
```

```
       loserscore  loseroldrating  losernewrating  loserpos
round  division  \
77              0            1209            1188        23
5         2
78              0            1307            1282        17
6         2
137             0            1209            1250        13
4         2
231             0            1401            1310        24
7         2
1138            0            1416            1361        61
1         2
...           ...             ...             ...       ...
...       ...
95751           0            1424            1447         6
7         2
95799           0            1485            1475        14
7         2
96107           0             977             999         9
7         2
96801           0            1449            1446         9
1         2
98698           0            1179            1157        21
7         2

             date  lexicon
77     1998-12-06    False
78     1998-12-06    False
137    1998-12-06    False
231    1998-12-06    False
1138   1999-01-22    False
...           ...      ...
95751  2000-11-11    False
95799  2000-11-11    False
96107  2000-11-04    False
96801  2000-11-05    False
98698  2000-11-11    False

[139 rows x 19 columns]
```

In the above example, we have selected those games that ended in a tie in the second division.

6.2.10

Load the data from the dataset titanic.csv. Examine the data in the dataset and use the **loc** function to find out how many underage boys (<18) were on the Titanic. Don't forget to solve for the missing values first! Since we can't fill in the ages, it will be convenient to delete the missing values.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

6.2.11

Load the data from the dataset titanic.csv. Examine the data in the dataset and use the **loc** function to find out how many Titanic passengers survived and were between the ages of 20-40?

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

6.2.12

There are two ways to access DataFrame data, either by notation or by notation. One option, already presented, is using **square brackets**. This option also has multiple approaches:

- selection of the entire column:
- *df['column']* - the result is a series
- *df[[list of columns]]* - the result is a DataFrame
- row range selection:
- *df[start:end]* - start and end are positive integers and the result is a DataFrame
- *df[start:end:step]* - start, end are positive integers, step can also be a negative number that denotes every n-th row, which will be in the resulting DataFrame

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df['winnername'])
print(df[20:30:2])
```

The second option uses the so-called **dot** notation. It is access to an attribute using a **dot** character. It can do similar things to square bracket notation but has some limitations. It is with the **loc** and **iloc** functions that we recommend using square brackets rather than dots. The biggest disadvantage of dot notation is that it uses a similar approach to attributes as Python, so if the column name is similar to any existing method or doesn't meet the variable naming standard (e.g. using diacritics), it will result in an error. For example, *df.max* will not work, so it is better to use the *df['max']* approach.

```
print(df.losername)
```

6.3 Adding a new column into DataFrame

6.3.1

In the following lessons, we will show how to manipulate the DataFrame. Let's start by saying that we want to add a new column to an existing DataFrame. To add new columns, we can use the **insert()** function, which updates the existing DataFrame. The other option, however, is to call the **assign()** function, which adds a new column and creates a new DataFrame. Let's first create a simple DataFrame representing this portal.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57]
}
df = pd.DataFrame(priscilla)
print(df)
# add new column
tests = [2,2,1,1,0]
df = df.assign(Tests=tests)
print(df)
```

Although we mentioned that the **assign()** function creates a new DataFrame, if we assign it to the original one, we overwrite it with the new content. We created a new column in the form of a list containing the number of tests in the given course and added it to the DataFrame. By analogy, we can also add multiple lists at once separated by a comma to the **assign(name1=list1, name2=list2)** function.

6.3.2

Another option is the **insert()** function, which works similarly to **assign()**, but we can specify where we want to create the new column. The function works with three

parameters: the position of the new column, the name of the new column, and a list of data.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57]
}
df = pd.DataFrame(priscilla)
tests = [2,2,1,1,0]
df.insert(3,'Tests',tests)
print(df)
```

Program output:

```
   Course  Chapters  Lessons  Tests
0    Java        17       63      2
1  Python        17       55      2
2     SQL         9       30      1
3    HTML         9       30      1
4       C        17       57      0
```

6.3.3

The last option is to create a column by notating **square brackets**. In principle, this is a simple assignment of values to a column, if the column does not exist in the DataFrame, it will be created. If it is already there, its data will be overwritten with the new ones.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57]
}
df = pd.DataFrame(priscilla)
tests = [2,2,1,1,0]
df['Tests'] = tests
print(df)
```

Program output:

```
   Course  Chapters  Lessons  Tests
0    Java        17       63      2
1  Python        17       55      2
2     SQL         9       30      1
3    HTML         9       30      1
```

```
4         C          17         57         0
```

6.3.4

Run the following code to initialize the DataFrame. Then add a new column called *Type*, which will contain the following list:

```
['cold','cold','hot','cold','hot','cold','cold']
```

The answer is the correct command to the job to create the column (accepts **assign**, **insert**, or create column via square brackets).

```
import pandas as pd
cereal = {
  'name': ['Apple Cinnamon Cheerios','Apple Jacks','Basic 4','Bran Chex','Bran Flakes',"Cap'n'Crunch",'Cheerios'],
  'manufactor': ['General Mills','Kelloggs','General Mills','Ralston Purina','Post','Quaker Oats','General Mills'],
  'calories': [110,110,130,90,90,120,110]
}
df = pd.DataFrame(cereal)
print(df)
# add a new column type
```

6.4 Removing data from DataFrame

6.4.1

We have already partially mentioned deleting data from the DataFrame in the chapter dealing with missing values. However, sometimes it can happen that we have too much information in the data file that we don't need. One option is to select only the essential data using the **loc** or **iloc** functions. The other option is to delete unnecessary columns. To delete data, there is a **drop()** function in pandas. By default, the function works by not deleting data from the current DataFrame but creating a new one. If we want to delete the columns of the current DataFrame, we use the **inplace=True** parameter. Another important parameter is **axis**, which has two values **0** or **1**, where **0** specifies deletion of **rows** and **1** indicates deletion of **columns**.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57],
  'Tests': [2,2,1,1,0]
```

```
}
df = pd.DataFrame(priscilla)
print(df)
# drop the column Lessons
df2 = df.drop(['Lessons'], axis=1)
print(df2)
```

Program output:

```
   Course  Chapters  Lessons  Tests
0    Java        17       63      2
1  Python        17       55      2
2     SQL         9       30      1
3    HTML         9       30      1
4       C        17       57      0
   Course  Chapters  Tests
0    Java        17      2
1  Python        17      2
2     SQL         9      1
3    HTML         9      1
4       C        17      0
```

The **drop()** function cleared the *Lessons* column from the DataFrame by passing it as the first parameter. An alternative is to specify it through a named parameter. Since we wanted to delete the column, we set **axis=1**. However, in order not to lose the column completely, we created a new DataFrame **df2**. If we wanted to, using the **inplace=True** command, we would overwrite the current DataFrame.

```
# alternative solution
df2 = df.drop(columns=['Lessons'], axis=1)
# inplace solution
df.drop(['Lessons'], axis=1, inplace=True)
```

6.4.2

Load the data from titanic.csv. Examine the data in the dataset and delete the column that contains the passenger ticket information from the dataset. The correct answer is the command to create that column (accepts assign, insert, or create column via square brackets to resolve). Use **df** as the name of the created DataFrame.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# drop the column
```

6.4.3

Another option is to delete a column based on a numeric index and call the column using the **columns** function. *It should not be forgotten that index numbering starts from 0*. Therefore, the following code will delete the second column.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57],
  'Tests': [2,2,1,1,0]
}
df = pd.DataFrame(priscilla)
print(df)
# drop the second column
df.drop(df.columns[[1]], axis=1, inplace=True)
print(df)
```

Program output:

```
   Course  Chapters  Lessons  Tests
0    Java        17       63      2
1  Python        17       55      2
2     SQL         9       30      1
3    HTML         9       30      1
4       C        17       57      0
   Course  Lessons  Tests
0    Java       63      2
1  Python       55      2
2     SQL       30      1
3    HTML       30      1
4       C       57      0
```

6.4.4

Load the data from titanic.csv. Examine the data in the dataset and delete the column that contains the passenger ticket price information from the dataset. Select all the correct alternatives to delete that column.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# drop the column
```

- `df.drop(df.columns[[9]],axis=1,inplace=True)`
- `df = df.drop(df.columns[[9]],axis=1)`
- `df.drop(['Fare'],axis=1,inplace=True)`
- `df = df.drop(['Fare'],axis=1)`

- `df = df.drop(['Fare'],axis=0)`
- `df.drop(['Fare'],axis=0,inplace=True)`
- `df.drop(df.columns[[8]],axis=1,inplace=True)`
- `df = df.drop(df.columns[[8]],axis=1)`

6.4.5

The **drop()** function can also delete more than one column. Again, we have several alternatives to write the deletion, either by calling the column names directly or via indexes. We can also send a list of columns directly as a parameter.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57],
  'Tests': [2,2,1,1,0]
}
df = pd.DataFrame(priscilla)
print(df)
# drop multiple column
df.drop(df.columns[[1,2]], axis=1, inplace=True)
print(df)
# alternatives
#df.drop(['Chapters','Lessons'], axis=1, inplace=True)
#listOfCols = ['Chapters','Lessons']
#df.drop(listOfCols, axis=1, inplate=True)
```

Program output:

```
   Course  Chapters  Lessons  Tests
0    Java        17       63      2
1  Python        17       55      2
2     SQL         9       30      1
3    HTML         9       30      1
4       C        17       57      0
   Course  Tests
0    Java      2
1  Python      2
2     SQL      1
3    HTML      1
4       C      0
```

6.4.6

As we have already shown in deleting columns, deleting rows is very similar because it works with the same **drop()** function. The main difference is in the **axis** parameter, which in the case of row deletion must be set to **0** (*since this is the*

default value of this parameter, we don't need to write it into the function). So we can delete rows in exactly the same way as we delete columns, using either row names or indexes. Rows are more likely to be denoted using indexes only and so you will encounter this option more often. In the following code, you can experiment with different options for deleting rows.

```
import pandas as pd
priscilla = {
  'Course': ['Java','Python','SQL','HTML','C'],
  'Chapters': [17,17,9,9,17],
  'Lessons': [63,55,30,30,57],
  'Tests': [2,2,1,1,0]
}
df = pd.DataFrame(priscilla, index=['r1','r2','r3','r4','r5'])
print(df)
# drop multiple rows
df.drop(df.index[[1,2]], inplace=True)
print(df)
# alternatives
#df.drop([0, 3], inplace=True)
#df.drop(['r1','r2'], inplace=True)
#df.drop(df.index[-1], inplace=True)
#
```

Program output:

```
    Course  Chapters  Lessons  Tests
r1    Java        17       63      2
r2  Python        17       55      2
r3     SQL         9       30      1
r4    HTML         9       30      1
r5       C        17       57      0
   Course  Chapters  Lessons  Tests
r1   Java        17       63      2
r4   HTML         9       30      1
r5      C        17       57      0
```

6.4.7

Load the data from titanic.csv. Examine the data in the dataset and delete the last 10 rows from the dataset. Select the correct notation to delete the given rows.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# drop the rows
```

- `df.drop(df.index[-10:],inplace=True)`

- `df.drop([-10:],inplace=True)`
- `df.drop(df.index[-10],inplace=True)`
- `df.drop([-10],inplace=True)`

6.4.8

In addition to deleting specific rows based on indexes or name tags, we can also delete rows based on a condition linked to a selected column. This is similar to deleting missing values but in this case, we can delete data we are not interested in based on the condition. We still use the **drop()** function and the first parameter is basically the condition of the selection of the data we want to delete. For example, we can remove the information about games that ended in a tie from the Scrabble tournament data, so we can focus only on wins and losses. In most cases, the alternative to deleting records is to select the data based on a similar condition, as we showed in the Data Selection chapter.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df.info())
# remove rows based on a condition
df.drop(df[df['tie']==True].index, inplace = True)
print(df.info())
```

We deleted all records from the data file that had a tied value of *True*. We can similarly delete based on more complex conditions, using the logical operators **and** or **or**. Remember to enclose each condition in parentheses. In the following example, we can select only those games where the winner significantly beat the opponent.

```
df.drop(df[(df['winnerscore']>400) &
(df['loserscore']<200)].index, inplace=True)
# alternative data selection using loc
#df2 = df.loc[(df['winnerscore']>400) &
(df['loserscore']<200)]
print(df.info())
```

6.4.9

Load the data from titanic.csv. Review the data in the dataset and delete the passengers from the dataset so that only passengers who did not survive the disaster and were over 40 years of age **remain** in the data matrix. Pass the correct notation of the condition to the task, which will be placed in **df.drop()**.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# drop the rows
```

6.5 Working with data in DataFrame

6.5.1

As with other structures such as lists or dictionaries, we have the ability to loop through the DataFrame row by row accessing a specific column of each row. Pandas uses the **iterrows()** or **itertuples()** functions for searching.

The **iterrows()** function is used to scan rows, which returns as a result (index, series), where index is the row index and series contains the data from each column for that row. In order to get the data for a particular column, we need to access the series e.g. **row['column']**.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
for index, row in df.iterrows():
  print(index, row['Date'], row['Water'])
```

Program output:

```
0 2020-01-31 13.0
1 2020-02-29 18.2
2 2020-03-31 nan
3 2020-04-30 18.8
4 2020-05-31 14.2
5 2020-06-30 8.4
6 2020-07-31 16.4
7 2020-08-31 7.800000000000001
8 2020-09-30 2.4
9 2020-10-31 9.6
10 2020-11-30 16.599999999999998
11 2020-12-31 8.4
12 2021-01-31 17.6
13 2021-02-28 7.0
14 2021-03-31 2.4
15 2021-04-30 2.4
16 2021-05-31 9.2
17 2021-06-30 11.4
18 2021-07-31 8.0
19 2021-08-31 0.8
```

```
20 2021-09-30 13.4
21 2021-10-31 9.2
22 2021-11-30 13.2
23 2021-12-31 4.8
```

6.5.2

The second option is to search using the **itertuples()** function, which is the most commonly used, as it returns all DataFrame elements in the form of an **iterator** that contains **tuples** for each row. Also, in terms of DataFrame processing speed, the **itertuples()** function is faster than **iterrows()**. The **itertuples()** function has one essential parameter - **index**, which can have two *True/False* states and corresponds to whether we want our tuples to contain the row index or not. The default state is *True* and therefore if we call the function without parameters, the tuples will automatically contain the row index. The second parameter is **name**, which we can use to set the name of the tuples.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
for row in df.itertuples():
  print(row)
```

As we can see, the content of the **row** variable is a tuple, so if we want to access the individual columns, we use the following notation using the **getattr()** function.

```
for row in df.itertuples():
    print(getattr(row,'Index'),getattr(row, "Household"),
getattr(row, "Gas"))
```

6.5.3

In other programming languages, we often encounter the **for** loop when searching through various one- and two-dimensional structures. We can also use it in the case of DataFrame searches but from the time point of view, this method is the most time-consuming. Therefore, this kind of search is not recommended for larger data sets. We can use the row index as the control variable of the loop.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
for index in df.index:
  print(df['Household'][index], df['Gas'][index])
```

We get a similar result if we use auxiliary variable *i*, which will take values from the DataFrame length range and use the **loc** or **iloc** function to retrieve the data.

```
# loc
for i in range(len(df)):
  print(df.loc[i,'Household'], df.loc[i,'Gas'])
# iloc
for i in range(len(df)):
  print(df.iloc[i,1], df.iloc[i,4])
```

6.5.4

The last option is a bit more difficult because we will not use any loop directly but we will use the **lambda** function above the **apply()** function. A lambda function is basically a simple one-line anonymous function. And it is the **apply()** function that is used to support running functions over DataFrame rows.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
print(df.apply(lambda row: str(row['Household'])+"
"+str(row['Waste']), axis = 1))
```

Program output:

```
0      1748.0 nan
1      1748.0 nan
2         nan nan
3      1748.0 nan
4      1748.0 nan
5      1748.0 nan
6      1748.0 nan
7      1748.0 8.0
8      1748.0 3.0
9         nan 4.0
10     1748.0 7.0
11     1748.0 3.0
12     1748.0 3.0
13     1748.0 3.0
14     1748.0 6.0
15     1748.0 7.0
16     1748.0 2.0
17     1748.0 6.0
18     1748.0 5.0
19     1748.0 1.0
20     1748.0 2.0
```

```
21     1748.0 5.0
22     1748.0 8.0
23     1748.0 7.0
dtype: object
```

6.5.5

Load the data from titanic.csv. There were an estimated 2,224 passengers aboard the Titanic. Examine the data in the dataset and see how many passengers we have recorded. List the proportion of passengers accounted for and what percentage of them survived. Write the result as a percentage and round to whole numbers, keeping the following notation:

```
80% pas, 93% sur
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

6.5.6

Load the data from titanic.csv. Examine the data in the dataset to see how many passengers died and had a ticket on the lower deck. This information can be found in the Pclass column, where 1 is upper, 2 is middle and 3 is the lower deck.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

6.5.7

Load the data from titanic.csv. Since the disaster happened quite a while ago, some data were only estimated - for example, the age of the passengers. Examine the data in the dataset and see how many passengers had their ages estimated. If the age was estimated, it was encoded as a decimal number xx.5. However, do not consider ages less than 1, as this was set as a fraction.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

Data Summarization

Chapter 7

7.1 Data grouping

7.1.1

Data summarization is one of the most essential tasks in data analysis because, in this step, the data analyst converts a large amount of data into a few primary data summaries. Therefore, we will first discuss how to group and aggregate the data and then look at the basic statistics capabilities of the pandas library.

In general, datasets consist of one observation per row, which means we can get datasets containing millions of rows. Of course, deriving any data analysis based on tens of rows is not the same as millions of rows. In such situations, grouping/summarizing rows based on common variables is a good solution. For grouping, the **groupby()** function is used in pandas. The result of the function call is the DataFrameGroupBy structure, which provides us with several aggregation functions such as **sum()**, **mean()**, **median()**, and so on.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_games.csv', sep=',')
df2 = df.groupby(['winnername']).sum()
print(df2)
```

The above code grouped our data based on the winners of the Scrabble duels, summing the individual numeric columns. However, as we can see, it also summed columns that only contained, for example, the tournament identifier, which is not appropriate.

7.1.2

Load the data from titanic.csv. Examine the data in the dataset and use gender grouping to determine the number of male survivors of the disaster.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

7.1.3

We often encounter the need to aggregate data for more than one column. We follow the same procedure as in the previous cases and send a list of columns to the function. If we don't want to do grouping for all columns, we can select a group of columns of interest to us as groupby. If we assign the grouped DataFrame to a variable, we can then call aggregate functions over that data.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
df2 = df.groupby(['date', 'round'])[['winnerscore']]
print(df2.mean())
```

In this case, we aggregated the data for each day and the rounds played on that day and we were interested in the average winning score per round.

7.1.4

Load the data from titanic.csv. Examine the data in the dataset and using clustering based on gender and deck (Pclass, where 1 is upper, 2 is middle, and 3 is lower deck), find the average age of males on the lower deck.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

7.1.5

When we use the grouping function to create a new structure, the default state is that this new structure does not contain an index for the rows. Respectively, even if it is there, it is still original and the values do not follow one by one. Therefore, when creating grouped data, it is possible to call the **reset_index()** function, which resets the index for the new dataset and renumbers it again from 0.

Another important feature of the **group_by()** function is the **dropna** parameter, which defaults to *True*, meaning that it will not consider missing values in the aggregation. If we would like to include them, we need to set this parameter to *False*.

The last parameter we'll mention is **sort**, which also defaults to *True*. This means that the values are automatically sorted by the grouped column in ascending order. This is a time-consuming operation, but we may sometimes need to turn it off even if we want the reverse ordering.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
df2 = df.groupby(['date', 'round'],
sort=False)[['winnerscore']].mean().reset_index()
sorted_df = df2.sort_values(['date','round'], ascending=False)
```

```
print(sorted_df)
```

Program output:

```
            date  round  winnerscore
2224  2000-12-31      7    54.133333
2226  2000-12-31      6    55.533333
2223  2000-12-31      5    57.266667
2222  2000-12-31      4    51.733333
2225  2000-12-31      3    51.933333
...          ...    ...          ...
8     1998-12-06      5     0.000000
3     1998-12-06      4     0.000000
2     1998-12-06      3     0.000000
1     1998-12-06      2     0.000000
0     1998-12-06      1     0.000000

[2241 rows x 3 columns]
```

7.1.6

We've introduced a few aggregation functions, but we've only ever used one. Pandas allows us to use the **aggregate()** function to send a list of aggregation functions that we want to get over an aggregated column. We can call the following aggregate functions:

- count() - returns the count of group elements
- size() - returns the size of each group
- sum() - returns the total sum of each group
- mean() - returns the average of each group (same as average())
- average() - returns the average of each group (same as mean())
- std() - returns the standard deviation of each group
- describe() - returns various basic statistics
- min() - returns the minimum of each group
- max() - returns the maximum of each group
- first() - returns the first value of each group
- last() - returns the last value of each group

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
df2 = df.groupby(['date',
'round'])[['winnerscore']].aggregate(['min','max'])
print(df2)
```

Program output:

```
                 winnerscore
                         min  max
```

```
date        round
1998-12-06  1            0    0
            2            0    0
            3            0    0
            4            0    0
            5            0    0
...                    ...  ...
2000-12-31  3            0  395
            4            0  392
            5            0  440
            6            0  436
            7            0  425

[2241 rows x 2 columns]
```

7.1.7

Load the data from titanic.csv. Examine the data in the dataset and use the grouping and aggregate functions to find the youngest and oldest passengers by gender. Write the solution in the form (keep the decimal numbers returned by the dataset):

```
youngest_female, youngest_male, oldest_female, oldest_male
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore dataset
```

7.1.8

Load the data from titanic.csv. Examine the data in the dataset and use the grouping and aggregate functions to see which port had the most passengers boarding. In the dataset, this information is in the Embarked column, where the abbreviations represent the following ports:

- C = Cherbourg,
- Q = Queenstown,
- S = Southampton.

Write the name of the port and the number of passengers who boarded in the following format:

```
port: number
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
```

```
# explore dataset
```

7.1.9

The last function we will mention in this chapter is the **unique()** function, which allows us to get unique values from columns. The function is applied to a series, so we must always select a specific column over which to call the function. The result is a list of unique values sorted by occurrence.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
winners = df['winnername'].unique()
print(winners)
```

Program output:

```
['Harriette Lakernick' 'Patricia Barrett' 'Sam Dick-Onuoha'
...
 'Monica Disponett' 'Dixie Davis' 'Jason Allain']
```

7.1.10

Load the data from titanic.csv. Examine the data in the dataset and find out how many unique cabins were on the Titanic. In the dataset, this information is in the Cabin column. Be sure to check for missing data.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

7.2 Pivot tables

7.2.1

A pivot table is a very effective tool to better understand the data. It can summarize and organize large data from a larger table. A pivot table can contain totals, averages, and various other statistics that are grouped together in a meaningful way. You've probably already encountered it in a spreadsheet calculator, such as Excel. Similarly, we can easily create a contingency table using the pandas library. The **pivot_table()** function works on a similar principle as the **groupby()** function, which is used to group data.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
table = pd.pivot_table(df, index=['division','round'])
print(table)
```

Above we have created a contingency table for each division and round of play in Scrabble tournaments. As we can see, all the numerical columns have been entered into the table, and the averages for each round for specific divisions have been automatically created. The advantage of contingency tables is that it is really quick to visualize the data in spreadsheet form. The **index** parameter gives us the grouping condition for the table. We have the option to add another dimension to the table using the **columns** parameter, which creates an additional level grouping.

```
table2 = pd.pivot_table(df, index=['division','round'],
columns=['tourneyid'])
print(table2)
```

Program output:

```
                 gameid                                              
\
tourneyid             1           2        3        4             5  
6
division round                                                         
1        1         76.4  289.000000  658.00  1635.6  1828.666667  
1970.666667
         2         98.2  296.000000  674.55  1626.6  1817.500000  
1987.333333
         3         89.8  293.166667  691.15  1638.8  1817.000000  
1973.666667
         4         48.8  298.000000  698.25  1603.8  1789.000000  
1977.000000
         5         81.0  292.666667  762.75  1611.2  1824.333333  
1974.666667
...                 ...         ...      ...      ...          ...  
...
9        15         NaN         NaN      NaN      NaN          NaN  
NaN
         16         NaN         NaN      NaN      NaN          NaN  
NaN
         17         NaN         NaN      NaN      NaN          NaN  
NaN
         18         NaN         NaN      NaN      NaN          NaN  
NaN
```

```
         19        NaN          NaN      NaN      NaN            NaN
NaN

                                                              ...
winnerscore       \
tourneyid           7       8             9        10     ...
338  339
division round                                            ...
1        1       2050.25  2152.0  2244.666667  2344.50  ...
0.0  0.0
         2       2055.75  2144.5  2256.000000  2346.75  ...
0.0  0.0
         3       2063.50  2151.0  2249.000000  2390.25  ...
0.0  0.0
         4       2076.75  2154.0  2254.000000  2398.50  ...
0.0  0.0
         5       2075.25  2150.5  2222.666667  2364.00  ...
0.0  0.0
...                 ...     ...           ...       ...   ...
...  ...
9        15          NaN     NaN           NaN       NaN  ...
NaN  NaN
         16          NaN     NaN           NaN       NaN  ...
NaN  NaN
         17          NaN     NaN           NaN       NaN  ...
NaN  NaN
         18          NaN     NaN           NaN       NaN  ...
NaN  NaN
         19          NaN     NaN           NaN       NaN  ...
NaN  NaN


tourneyid                 340  341      342  343  344  345
346  347
division round
1        1       60.166667  0.0  102.375  0.0  0.0  0.0
50.375000  0.0
         2       80.333333  0.0  105.000  0.0  0.0  0.0
100.555556  0.0
         3       75.166667  0.0  104.250  0.0  0.0  0.0
86.555556  0.0
         4       78.166667  0.0  104.500  0.0  0.0  0.0
86.222222  0.0
```

```
         5      74.000000  0.0  103.625  0.0  0.0  0.0
95.444444  0.0
...                    ...  ...      ...  ...  ...  ...
...  ...
9        15            NaN  NaN      NaN  NaN  NaN  NaN
NaN  NaN
         16            NaN  NaN      NaN  NaN  NaN  NaN
NaN  NaN
         17            NaN  NaN      NaN  NaN  NaN  NaN
NaN  NaN
         18            NaN  NaN      NaN  NaN  NaN  NaN
NaN  NaN
         19            NaN  NaN      NaN  NaN  NaN  NaN
NaN  NaN

[243 rows x 4472 columns]
```

7.2.2

Load the data from titanic.csv. Examine the data in the dataset and determine which group of passengers (male/female) by deck type (Pclass) was the youngest and what was the average age of that group by deck. Write the result in the following form, write the deck with the number as in the dataset:

```
gender, deck, average age
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

7.2.3

Contingency tables give us more options than simple grouping but of course, as we have already mentioned, the idea is similar. We've already seen the **index** parameter, which is used to specify the columns for which we will group the DataFrame data. Another important parameter is **values**, which specifies the values for which we want to create the contingency table.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
table = pd.pivot_table(df, index=['division','round'],
values='winnerscore')
```

```
print(table)
```

Program output:

```

```

7.2.4

Load the data from titanic.csv. Examine the data in the dataset and determine which group of passengers had the highest and lowest survival rates. We are interested in the combination of the port they boarded at and which deck they sailed on. The individual ports (Embarked) have the following codes:

- C = Cherbourg,
- Q = Queenstown,
- S = Southampton.

Decks (Pclass) are also represented by a code:

- 1 = upper deck (write upper),
- 2 = middle deck (write middle),
- 3 = under deck (write under).

Write the result in the following form, round the survival rate to whole numbers and write it as a percentage:

```
max: port, deck, % survival rate; min: port, deck, % survival rate
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

7.2.5

So far, we were still working with the default state of the contingency table, which only computed the average for the values we examined. We mentioned that we can do functions similar to the groupby and therefore **aggfunc** is an important parameter to specify aggregate functions. The notation of the aggregate functions is the same as in the case of clustering, with *sum*, *min*, *max*, *mean* being among the most commonly used. If we want the contingency table to compute multiple functions we have to write them as a list in the parameter.

The final step to complete the table is to add a summary row to Total. Depending on the aggregate function selected, the summary row will give us the information for the entire table and we set it via the **margins** parameter, which we set to *True*. This gives us an extra dimension to the data we are looking at, and especially if we still have a lot of information in the table, we can immediately see the overall

average or the smallest and largest value for the entire dataset. If we want, we can name the summary row using the **margins_name** parameter.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
df2 = df.loc[df['winnerscore']>0] # we look only at score > 0
table = pd.pivot_table(df2, index=['tourneyid','division'],
values='winnerscore', aggfunc=['mean','sum','min','max'],
margins=True, margins_name='Total')
print(table)
```

Program output:

```
                          mean         sum         min
max
                   winnerscore winnerscore winnerscore
winnerscore
tourneyid division
13         1        448.562500        7177         370
549
14         1        427.526316        8123         395
487
16         1        414.600000        6219         357
470
19         1        429.000000        8580         338
542
26         1        424.473684        8065         358
630
...                        ...         ...         ...
...
333        1        419.916667        5039         326
495
340        1        428.300000        8566         359
482
342        1        436.545455        9604         376
507
346        1        412.846154        5367         384
511
Total               404.495932     4922311           5
630

[73 rows x 4 columns]
```

7.2.6

Load the data from titanic.csv. Examine the data in the dataset and find out where the passenger who paid the highest amount for a ticket on Titanic boarded and which deck he was on. We recommend focusing on a combination of the port where they boarded and the ship they boarded. Evaluate the results for both genders. Also add up the total amount collected on tickets for all passengers.

The individual ports (Embarked) have the following codes:

- C = Cherbourg,
- Q = Queenstown,
- S = Southampton.

Decks (Pclass) are also represented by a code:

- 1 = upper deck (write upper),
- 2 = middle deck (write middle),
- 3 = under deck (write under).

Write the result in the following form, round the price of the tickets to whole numbers :

```
male: port, deck, price; female: port, deck, price; total:
price
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

7.3 Visualization

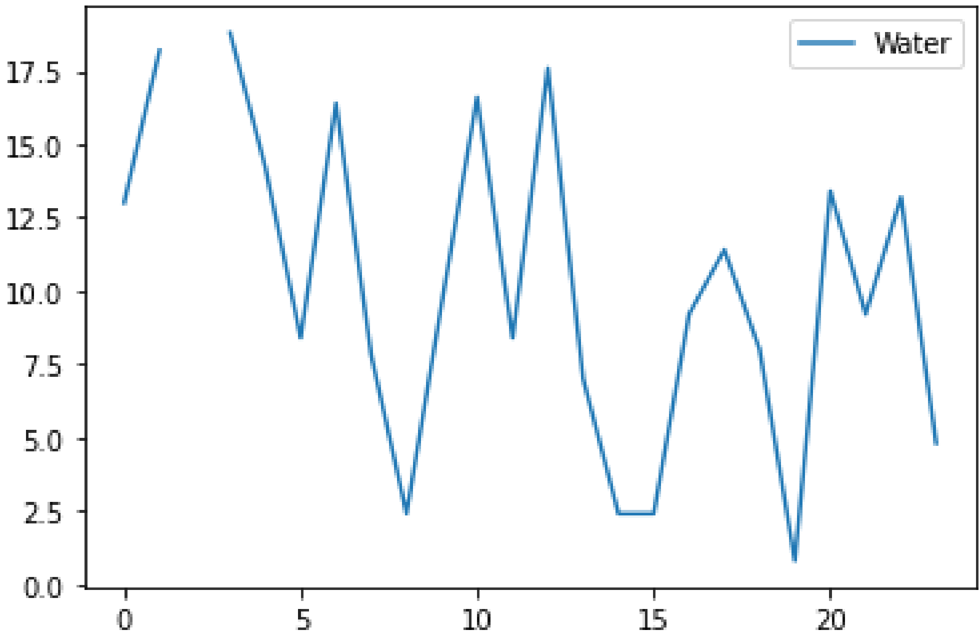
7.3.1

The pandas library offers various data visualization options in the form of generating graphs from DataFrames and series. This will help us to detect trends and relationships between different variables in our dataset. Pandas has built-in options to generate graphs in the form of **plot()** function. This function is linked to the popular visualization library **matplotlib** but pandas uses a simplified notation. This allows us to generate the same results with less code as with the original library.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
```

```
subdf = df[['Date','Water']]
subdf.plot()
```

Program output:



To ensure that our visualizations are accurate and that they correctly represent the knowledge extracted from the retrieved data, it is crucial to resolve missing data.

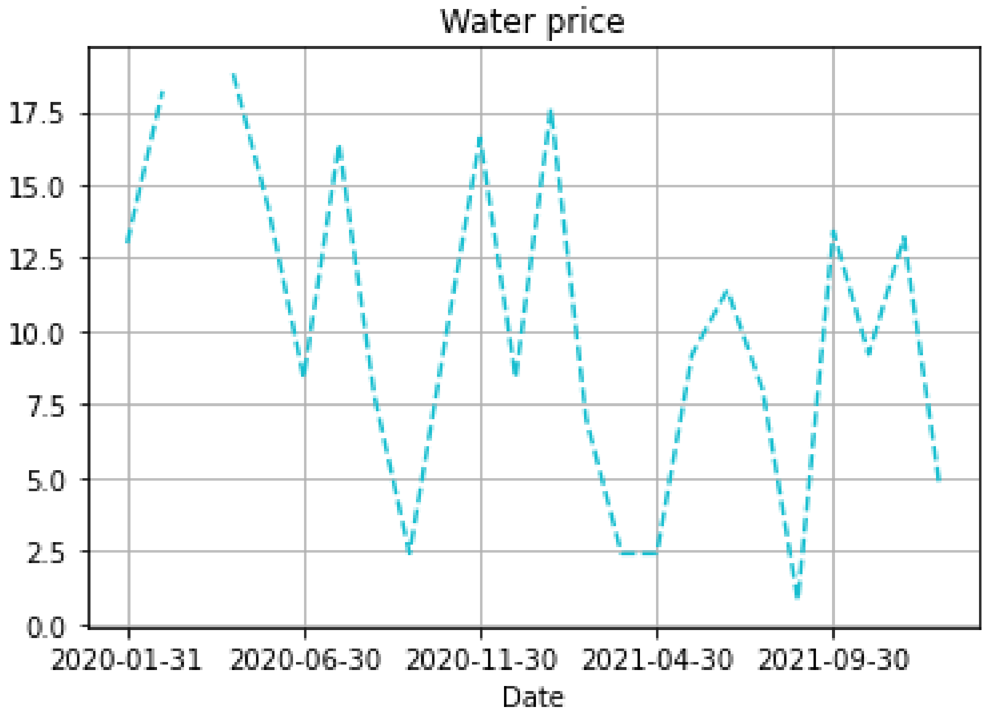
7.3.2

Of course, such simple graphs are not enough and it would be ideal to include essential information that will help the reader to better orient the displayed data. Therefore, we have several parameters that will enliven our graph. Let us imagine the functionality of the following parameters:

- **x**: represents the data on the x-axis, where we can specify the name of the column that is on the x-axis,
- **title**: we can create a caption for our chart,
- **legend**: is used to enable or disable the display of the legend in the chart (*True/False*),
- **grid**: is used to enable or disable the display of the grid in the chart (*True/False*),
- **color**: is used to change the color of a line in a chart, using the Tableau color palette (*tab:blue, tab:orange, tab:green, tab:red, tab:purple, tab:brown, tab:pink, tab:gray, tab:olive, tab:cyan*)
- **linestyle**: used to change the line style (*solid, dotted, dashed, dashdot*)

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
subdf = df[['Date','Water']]
subdf.plot(x='Date', title='Water price', legend=False,
grid=True, color='tab:cyan', linestyle='dashed')
```

Program output:



7.3.3

In addition to the visual aspect that completes the chart, the right type of chart is also essential. Not every chart type is suitable for the data being displayed. The evolution of water consumption over time can be shown in both bar and line graphs but a pie chart is less suitable. Conversely, a pie chart or bar chart is more appropriate for the age distribution of Titanic passengers. So how to decide which graph is appropriate for our data can be shown in the following paragraphs:

- if we want to compare the data, for example over time, we use a **line** graph,
- if we want to find out the composition of the data or its composition, we use a **pie** chart or an **area** chart,
- if we want to investigate the distribution of the data or its distribution, we use a **dot** plot or a **histogram**,
- if we want to investigate the relationship between the data, we use a **dot** or **bubble** plot.

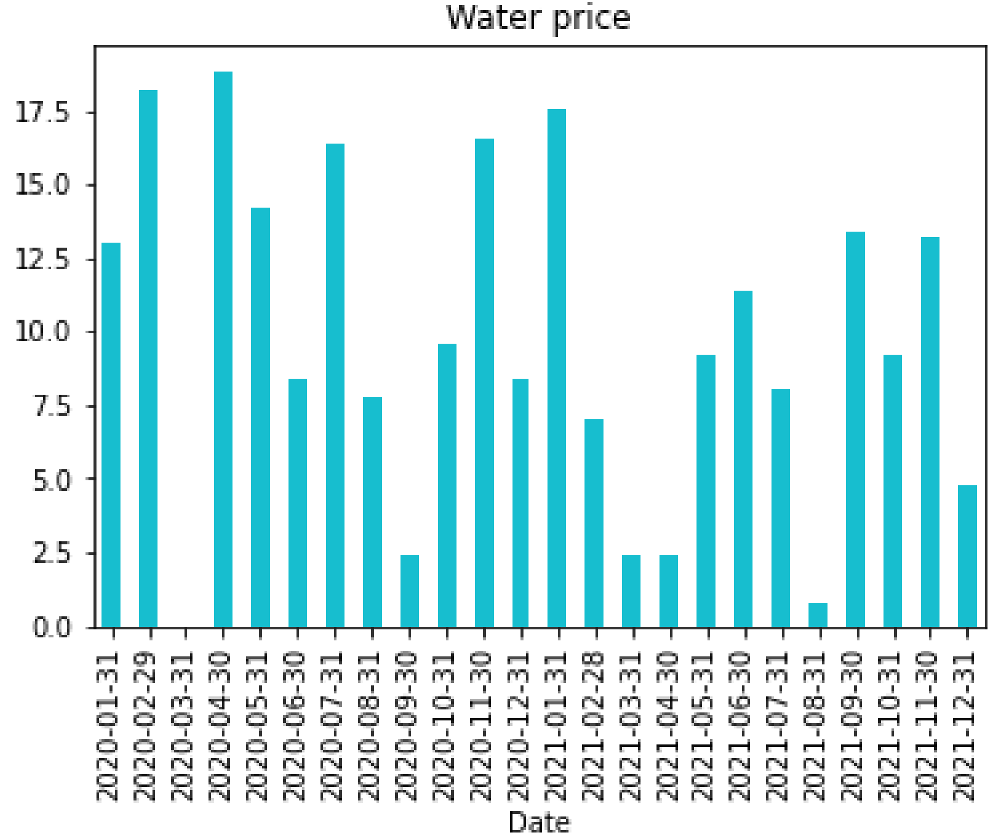
7.3.4

So once we know which data to use for which plot, we can specify it in the **plot()** function using the **kind** parameter. Again, we have several options for choosing the type of the plot:

- *line*: line graph (default option),
- *bar*: bar chart,
- *barh*: horizontal bar chart,
- *box*: box plot,
- *kde*: similar to histogram, density estimation plot,
- *density*: similar to kde,
- *area*: area graph,
- *pie*: pie chart,
- *scatter*: scatter plot,
- *hexbin*: a graph in the form of honeycombs (hexagons).

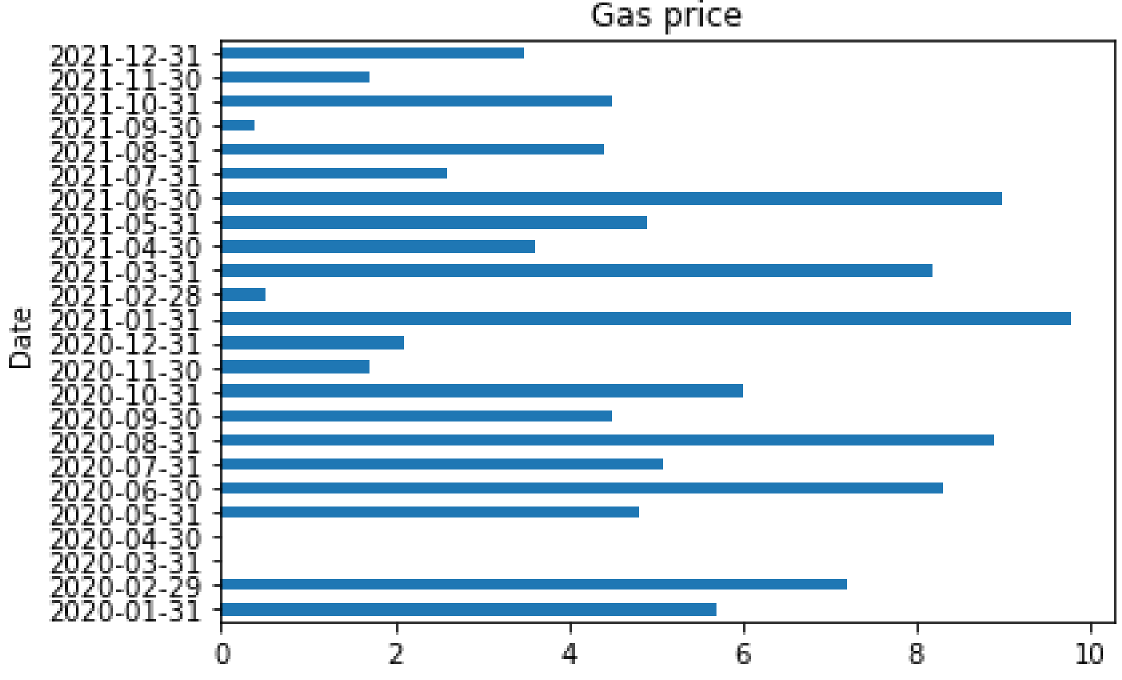
```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
subdf = df[['Date','Water']]
subdf.plot(x='Date', title='Water price', kind='bar',
legend=False, color='tab:cyan')
```

Program output:



```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
subdf = df[['Date','Gas']]
subdf.plot(x='Date', title='Gas price', kind='barh',
legend=False)
```

Program output:



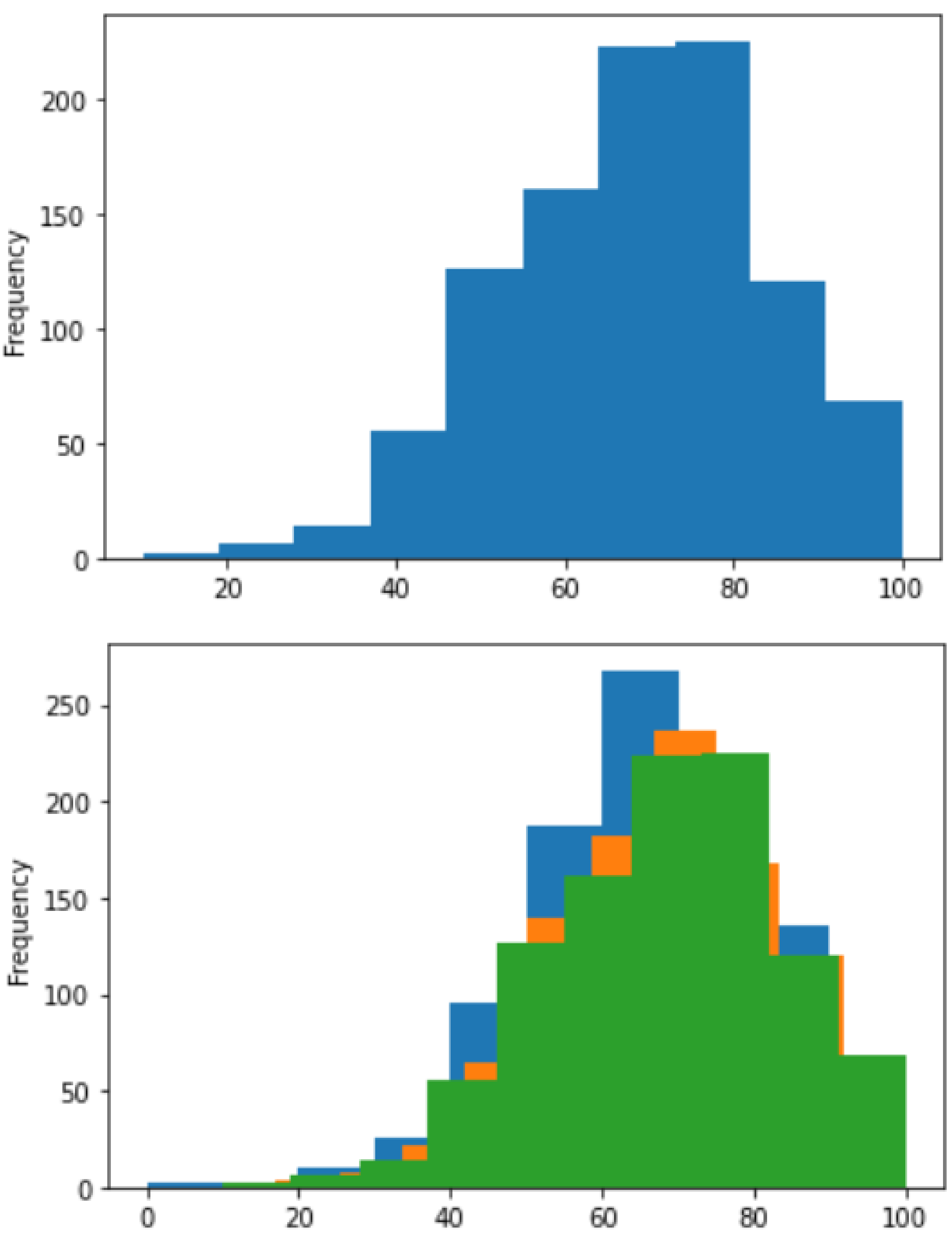

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
subdf = df[['Date','Electricty']]
subdf.plot(x='Date', title='Electricity price', kind='area',
legend=False, color='tab:red')
```

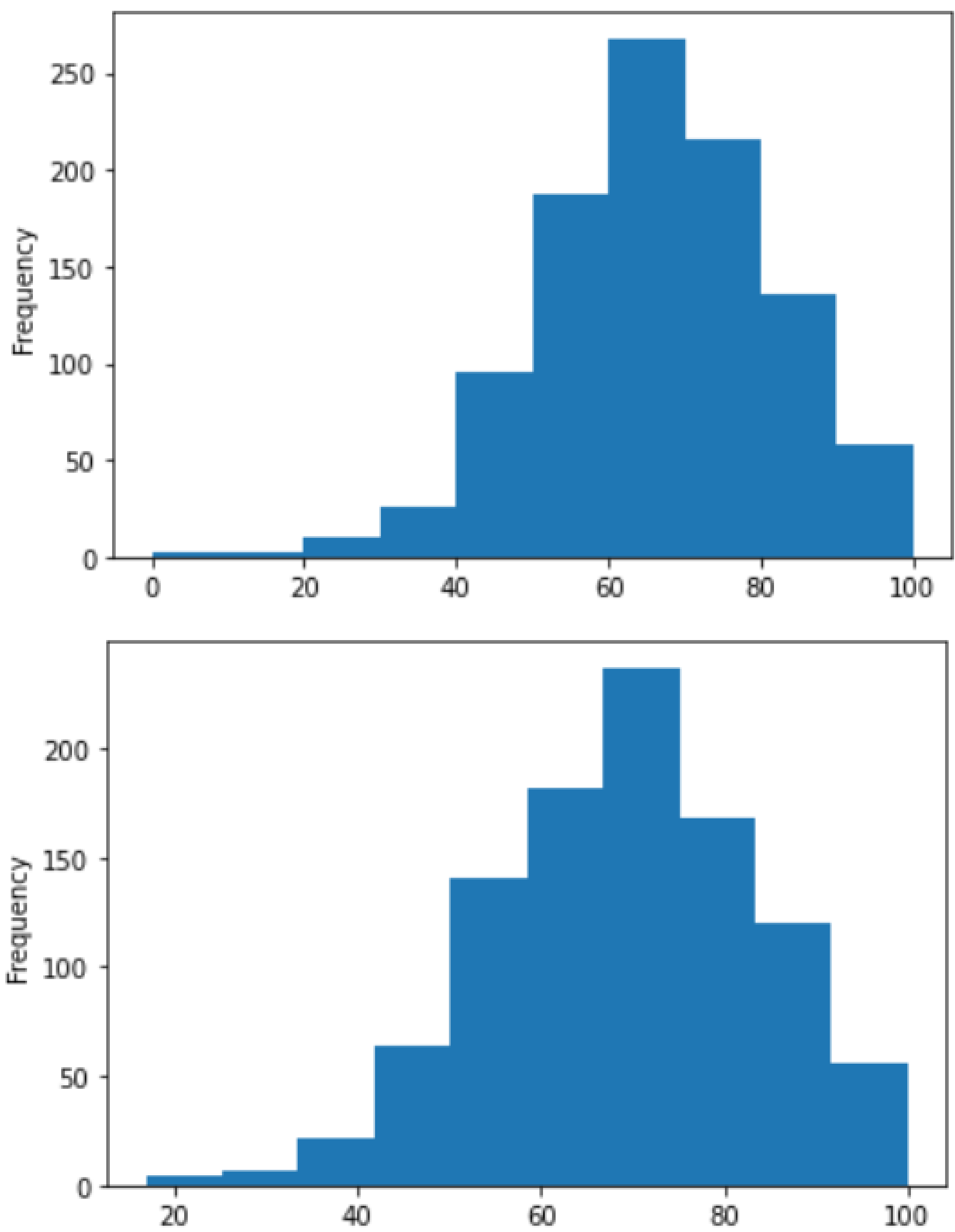
Program output:

7.3.5

Load data from StudentsPerformance.csv, which contains test results for math, reading, and writing. Explore the data in the dataset using the visualization to see which histogram visualizes the results from writing.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/StudentsPerformance.cs
v
# explore the dataset
```


Frequency
200
150
100
50
0
20
40
60
80
100
Frequency
250
200
150
100
50
0
0
20
40
60
80
100



7.3.6

Load the data from titanic.csv. Examine the data in the dataset using a histogram to see which age group was most represented on the ship. Write the result in the form lower limit - upper limit, for an interval of 10 years, e.g. 80-90.

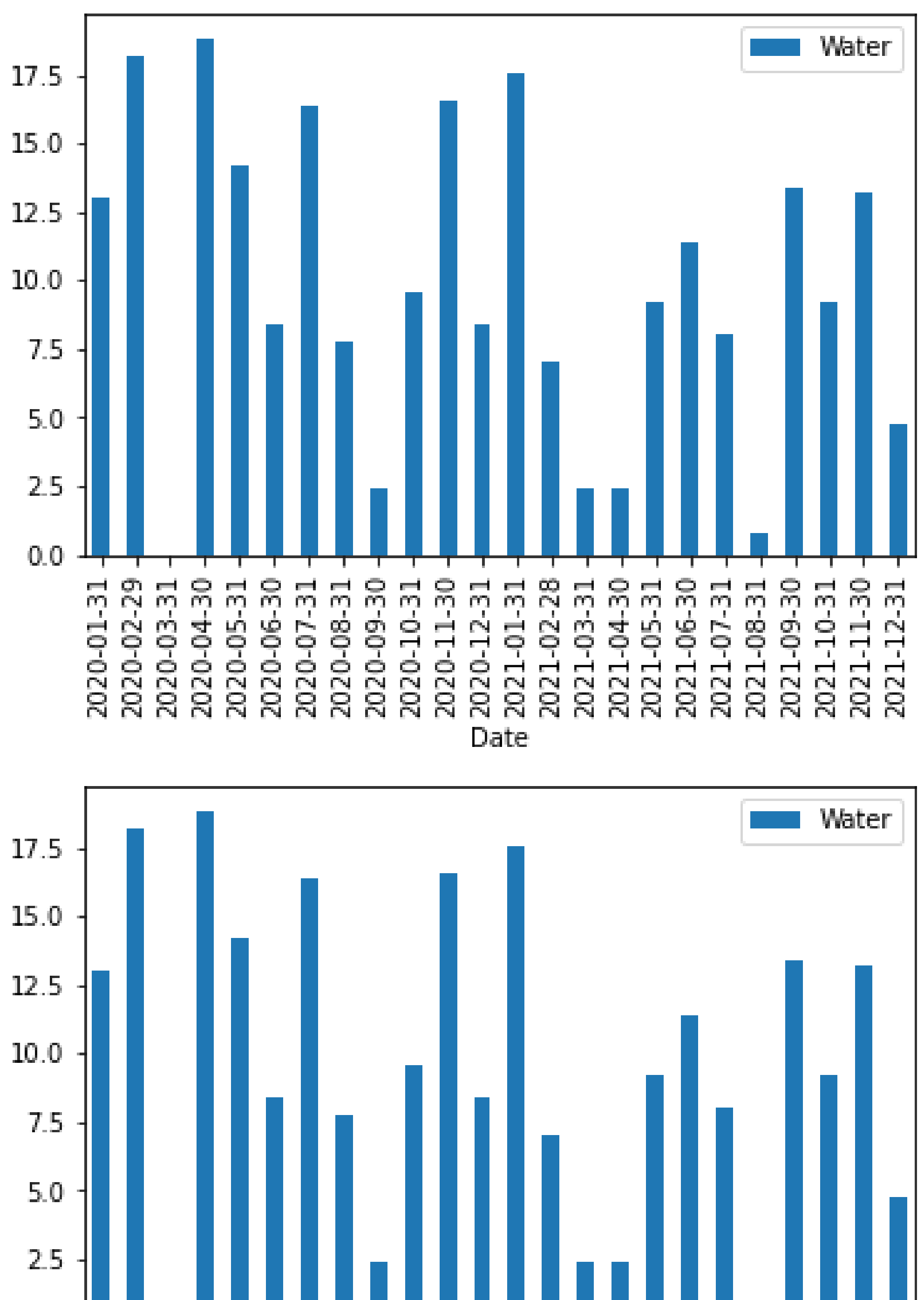
```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

7.4 Category variables

Categorical data refers to a type of data that can be divided into groups or categories. Only a fixed number of possible values and a limited grouping are considered categorical data. Examples of categorical data include nationality, sex, marital status, or occupation. Strings are not necessarily considered categorical data because categorical implies a sense of grouping. Names of people are strings but are not categorical because names are most likely to be unique, whereas age groups, such as 0-10 years, 11-20 years, and 21-30 years, are categorical because they represent groups. Sometimes, numerical data are grouped into small groups to form categorical data in order to get a better overview when analysing the data.

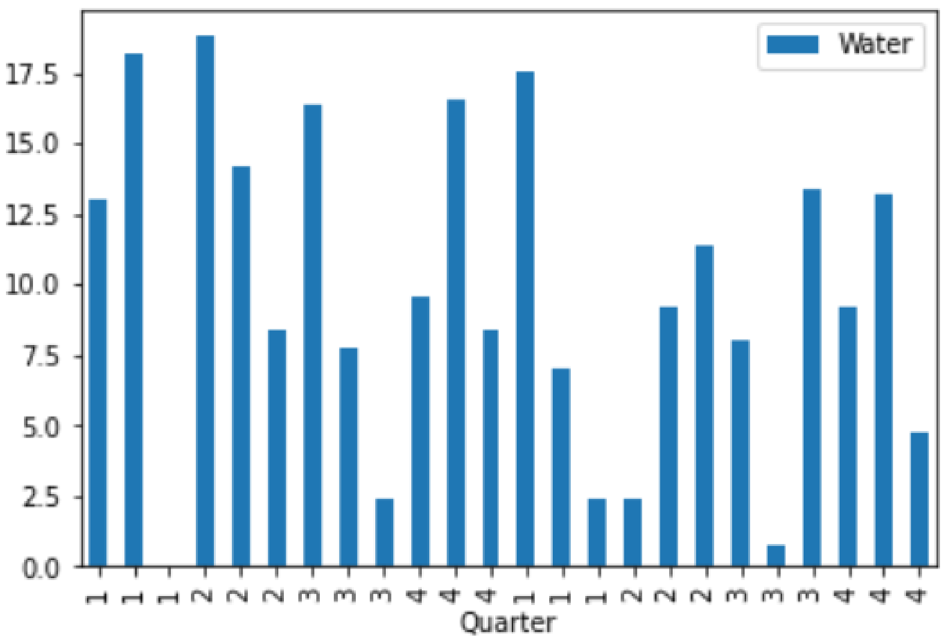
```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
df.plot(x='Date', y=['Water'], kind='bar')
# convert to datetime type
df.Date = pd.to_datetime(df.Date)
# create quarters from date
df["Quarter"] = df.Date.dt.quarter
df.plot(x='Quarter', y=['Water'], kind='bar')
```

Program output:


Water
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2020-01-31
2020-02-29
2020-03-31
2020-04-30
2020-05-31
2020-06-30
2020-07-31
2020-08-31
2020-09-30
2020-10-31
2020-11-30
2020-12-31
2021-01-31
2021-02-28
2021-03-31
2021-04-30
2021-05-31
2021-06-30
2021-07-31
2021-08-31
2021-09-30
2021-10-31
2021-11-30
2021-12-31
Date
Water
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
Quarter

7.4.2

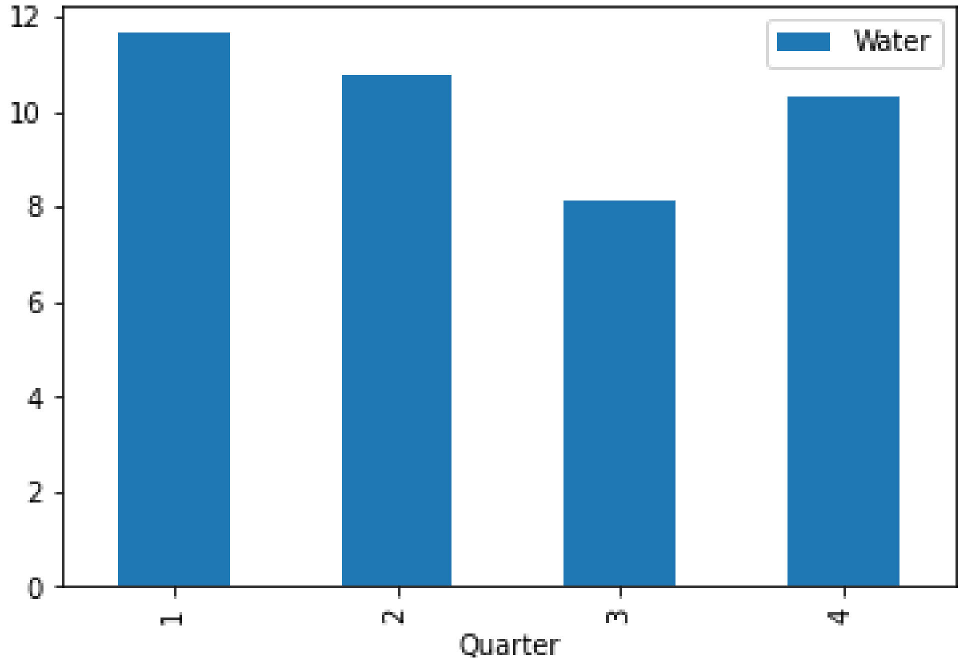
In the previous case, although we created quarters from dates, the data in the graph remained equally distributed.



Therefore, to get a better overview based on categories, we need to use some aggregation, such as an average. Then we can visualize a new DataFrame that contains the aggregated data.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
# convert to datetime type
df.Date = pd.to_datetime(df.Date)
# create quarters from date
df["Quarter"] = df.Date.dt.quarter
# agregate by Water
subdf = df.groupby(['Quarter'])[['Water']].mean()
subdf.plot(kind='bar')
```

Program output:



7.4.3

Another way to create a categorical variable is to use the **cut()** function. This function is used to divide the elements of a list into different categories, or **bins**. The parameters of the function are first a series of data from the DataFrame, followed by the **bins** parameter, which we use to define the condition for splitting the data. In our case, we divide the price of water consumed into three categories:

- 0< price <=10 - low
- 10< price <= 15 - normal
- 15< price <= infinity - high

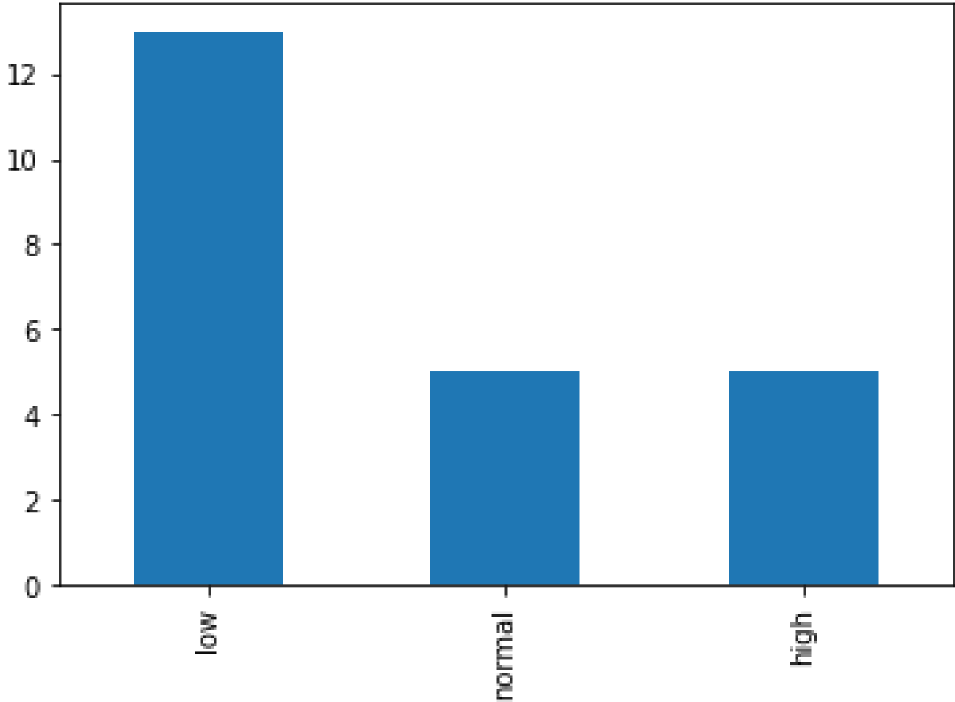
That condition containing equality is determined by the **right** parameter, which has a default value of *True*. The last parameter that is important to us is **labels**, which is used to create labels for each category. We can then visualize the frequency of occurrence of each category using the **value_counts()** function, which returns information about how many times the category occurs in the DataFrame. By connecting the **plot()** function, we can visualize the obtained counts directly.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/household
.csv', sep=';')
# create categories based on price
df['Water Prices'] = pd.cut(df['Water'],
bins=[0,10,15,float('Inf')], labels=['low','normal','high'])
print(df.head())
```

```
# visualize the counts of each category
df['Water Prices'].value_counts().plot(kind='bar')
```

Program output:

```
         Date  Household  Electricty  Water  Gas  Waste Water
Prices
0  2020-01-31     1748.0         7.0   13.0  5.7        NaN
normal
1  2020-02-29     1748.0        10.6   18.2  7.2        NaN
high
2  2020-03-31        NaN         NaN    NaN  NaN        NaN
NaN
3  2020-04-30     1748.0        18.6   18.8  NaN        NaN
high
4  2020-05-31     1748.0        12.0   14.2  4.8        NaN
normal
```



7.4.4

Load the data from titanic.csv. Examine the data in the dataset, focusing on the price group. Divide the ticket prices into intervals of 15 and determine which price group was the least numerous. You can put tickets with a price above $90 in the last group. Record the groups as follows:

```
0-15, 15-30, 30-45, 45-60, 60-75, 75-90, 90-...
```

```
# import library
```

```
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# explore the dataset
```

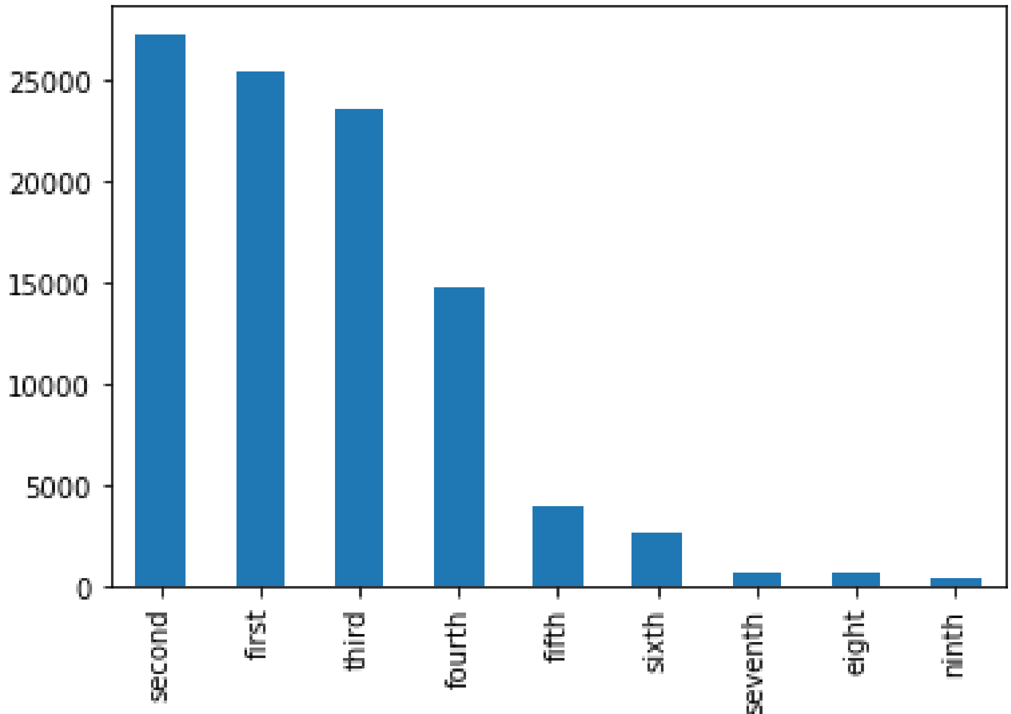
7.4.5

An alternative function to **cut()** is the **map()** function, which performs a similar function but works on a different principle. The **map()** function is used to perform a simple data transformation that uses a dictionary structure, where the *key* represents the "old" data and the *value* represents the new, transformed data. The use of this function is appropriate if we only want to convert numeric values to text values and thus obtain a categorical variable. Let's say that in our data set about Scrabble games, we want to examine individual divisions, so we can create categories based on the numeric values 1-9.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
print(df['division'].unique())
# map categories based on division
division_map = {1: 'first', 2: 'second', 3: 'third', 4:
'fourth', 5: 'fifth', 6: 'sixth', 7: 'seventh', 8: 'eight',
9:'ninth'}
df['cat_division'] = df['division'].map(division_map)
# visualize the counts of each category
df['cat_division'].value_counts().plot(kind='bar')
```

Program output:

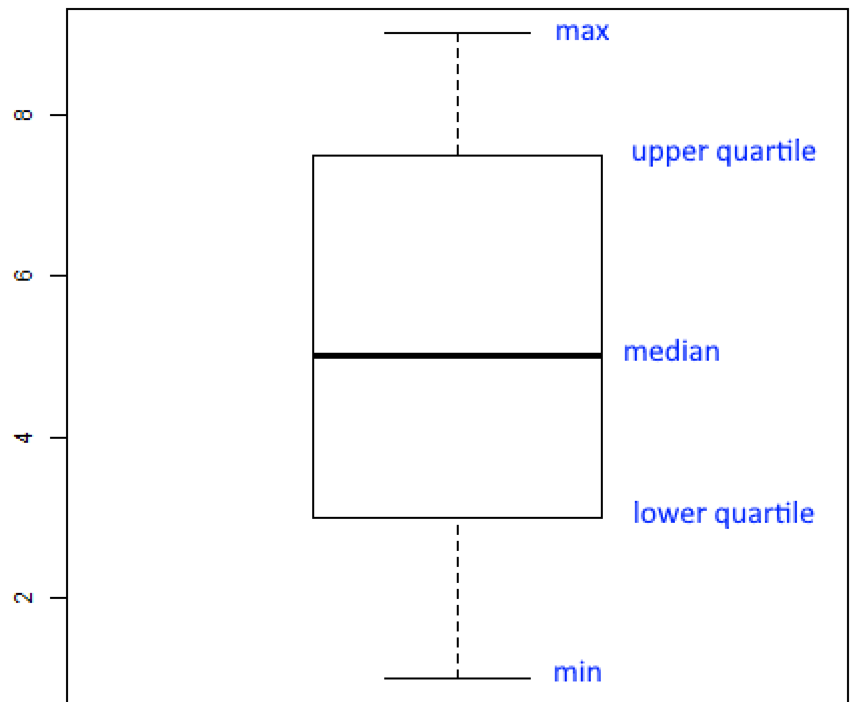
```
[1 3 2 4 5 6 7 8 9]
```



7.4.6

In the previous chapter, we focused on visualizations using the **plot()** function. Categorical variables give us a different way of looking at the visualization, and in this case *box plots* are often used. The pandas library supports special boxplots with the **boxplot()** function. First of all, however, let's focus on what the boxplot visualizes for us:

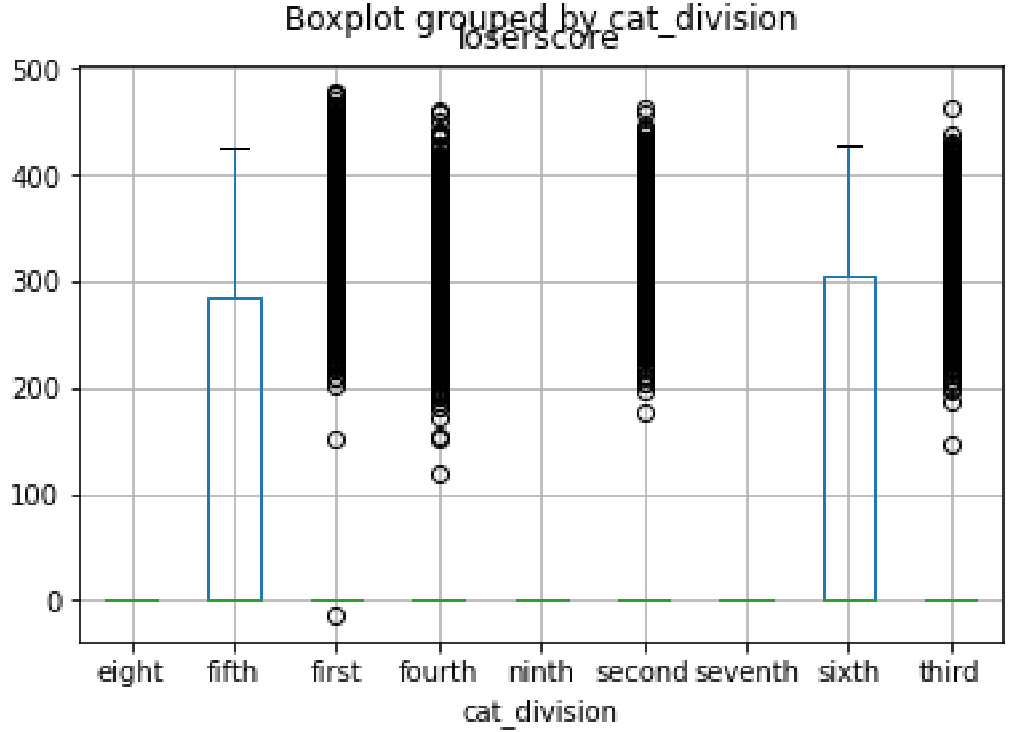
- the upper bound represents the highest value (max)
- the lower bound represents the lowest value (min)
- the middle value represents the median (median)
- the line between the upper limit and the median represents the upper quartile
- the line between the lower limit and the median represents the lower quartile



Working with the **boxplot()** function involves using the following parameters: **by** specifies, for example, a categorical variable and **column** specifies the variable we want to analyze or visualize. If we visualize from the dataset of Scrabble games the scores of the losers for each division, we can notice that in some divisions we don't have any graphs (1-4 and 7-9). This may indicate to us a problem with the dataset, or that this is clearly how the competitors in those divisions lost their games.

```
import pandas as pd
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/scrabble_
games.csv', sep=',')
# map categories based on division
division_map = {1: 'first', 2: 'second', 3: 'third', 4:
'fourth', 5: 'fifth', 6: 'sixth', 7: 'seventh', 8: 'eight',
9:'ninth'}
df['cat_division'] = df['division'].map(division_map)
# visualize the counts of each category
df.boxplot(by='cat_division', column=['loserscore'])
```

Program output:



7.4.7

Load the data from titanic.csv. Explore the data in the dataset using a **boxplot** to see what the distribution of age groups is across the different classes on the board. First, transform the Pclass variable into a categorical variable where the numbers represent the ship class: 1 = first, 2 = second, and 3 = third.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/titanic.csv
# create category Class
# visualize using boxplot by Class and Age
```

- the higher the class, the lower the age of the passengers
- the lower the class, the lower the age of the passengers
- the higher the class, the higher the age of the passengers
- the lower the class, the higher the age of the passengers

Project

Chapter 8

8.1 Spaceship Titanic - basic characteristics

The last chapter focuses on the application of the knowledge gained in the course over a similar dataset to the one you encountered in the assignments. In this case, we will be working with a Titanic-inspired dataset called "Spaceship Titanic". The dataset in question was created in order to compete in a model prediction task. However, our goal will only be to examine the dataset, which contains information about nearly 13,000 passengers who were moving from our solar system to three new planets. During the "voyage", the spacecraft encounters an anomaly and some of the passengers are transported to another dimension.

Consider the data file space_titanic.csv (the data is separated by a comma in the file) and the variables it contains:

- **PassengerId**: a unique identifier for each passenger in the format gggg_pp, where gggg represents the group of people they are travelling with and pp their group number,
- **HomePlanet**: the planet from which the passenger departed,
- **CryoSleep**: indicates whether the passenger has chosen to travel in deep sleep, if so the passenger is in his/her cabin,
- **Cabin**: the passenger's cabin number, written in the form deck/number/side, where side can be *p* for *port* and *s* for *starboard*,
- **Destination**: the planet to which the passenger is bound,
- **Age**: age of the passenger,
- **VIP**: information if the passenger has paid for VIP service during the cruise,
- **RoomService, FoodCourt, ShoppingMall, Spa, VRDeck**: variables informing how much money the passenger has spent in each luxury section of the ship,
- **Name**: first and last name of the passenger,
- **Transported**: information if the passenger has been transported to another dimension.

```
# import library
import pandas as pd
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
df =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/space_tit
anic.csv',sep=',')
# explore the dataset
print(df.head())
```

Program output:

```
  PassengerId HomePlanet CryoSleep  Cabin  Destination   Age
VIP  \
0      0001_01     Europa     False  B/0/P  TRAPPIST-1e  39.0
False
1      0002_01      Earth     False  F/0/S  TRAPPIST-1e  24.0
False
2      0003_01     Europa     False  A/0/S  TRAPPIST-1e  58.0
True
3      0003_02     Europa     False  A/0/S  TRAPPIST-1e  33.0
False
4      0004_01      Earth     False  F/1/S  TRAPPIST-1e  16.0
False

   RoomService  FoodCourt  ShoppingMall     Spa  VRDeck
Name  \
0          0.0        0.0           0.0     0.0     0.0
Maham Ofracculy
1        109.0        9.0          25.0   549.0    44.0
Juanna Vines
2         43.0     3576.0           0.0  6715.0    49.0
Altark Susent
3          0.0     1283.0         371.0  3329.0   193.0
Solam Susent
4        303.0       70.0         151.0   565.0     2.0  Willy
Santantines

   Transported
0        False
1         True
2        False
3        False
4         True
```

8.1.3

Load the data from space_titanic.csv. Examine the data in the dataset to see which variables **do not contain** missing data. As a result, write the variables separated with commas in the order they appear in the dataset. For example:

```
HomePlanet, Cabin, Age
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.4

Load the data from space_titanic.csv. Examine the data in the dataset and see what percentage of missing data each variable contains. Calculate first what percentage of missing values each variable contains and then you will be able to determine an approximate result for all variables. Round the result to whole numbers.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.5

Load the data from space_titanic.csv. Examine the data in the dataset and find out the average age of the passengers on the ship. Round the result to two decimal places.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.6

Read the data from space_titanic.csv. Examine the data in the dataset and see how many different groups of passengers are registered on the ship.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.7

Load the data from space_titanic.csv. Examine the data in the dataset and find out what percentage of passengers travel from Mars. Round the result to whole numbers.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.8

Load the data from space_titanic.csv. Examine the data in the dataset and find out what percentage of passengers travel in cryo-sleep. Round the result to whole numbers.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.9

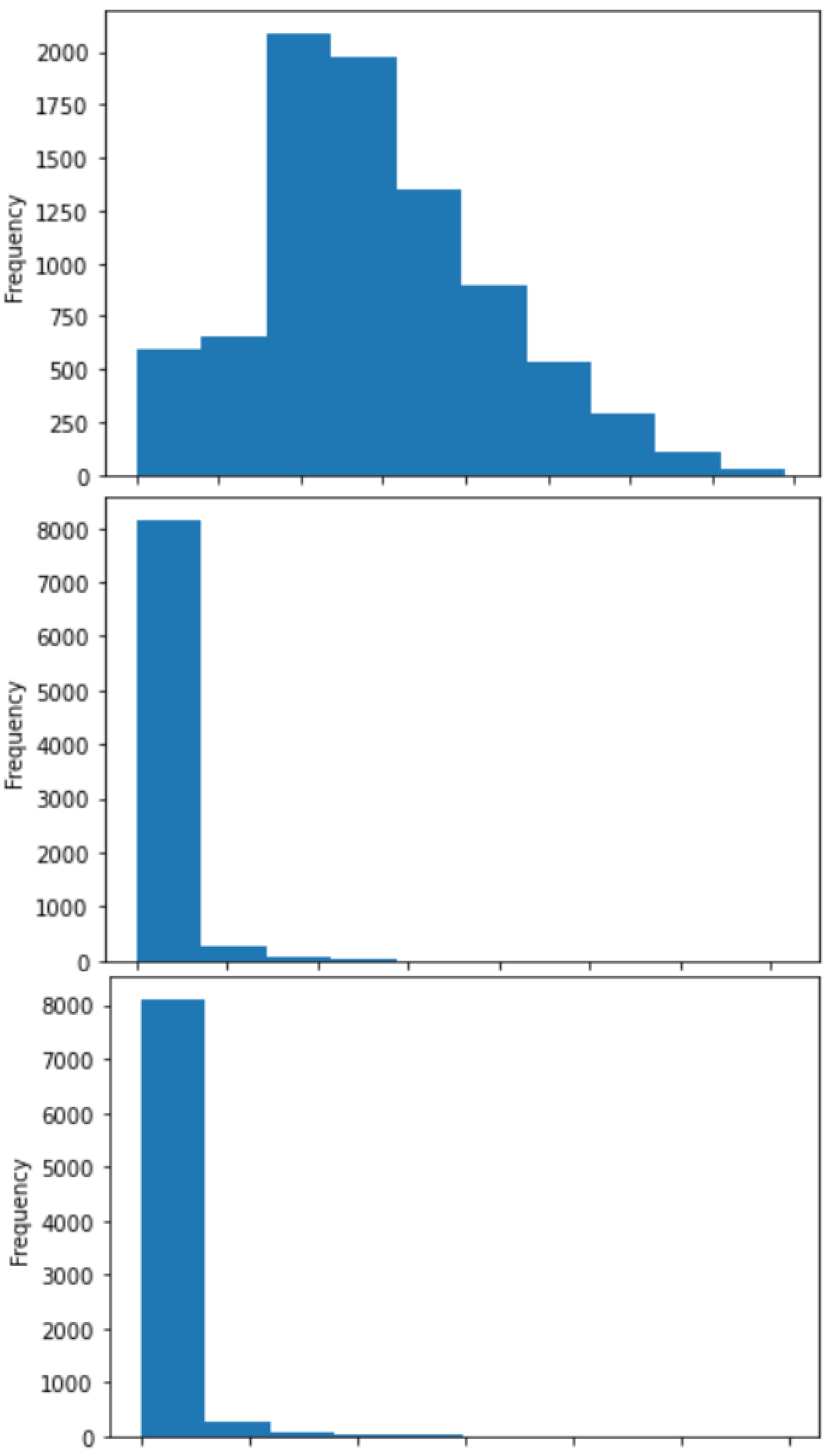
Load the data from space_titanic.csv. Examine the data in the dataset and find out what percentage of passengers paid for VIP services on the ship? Round the result to whole numbers.

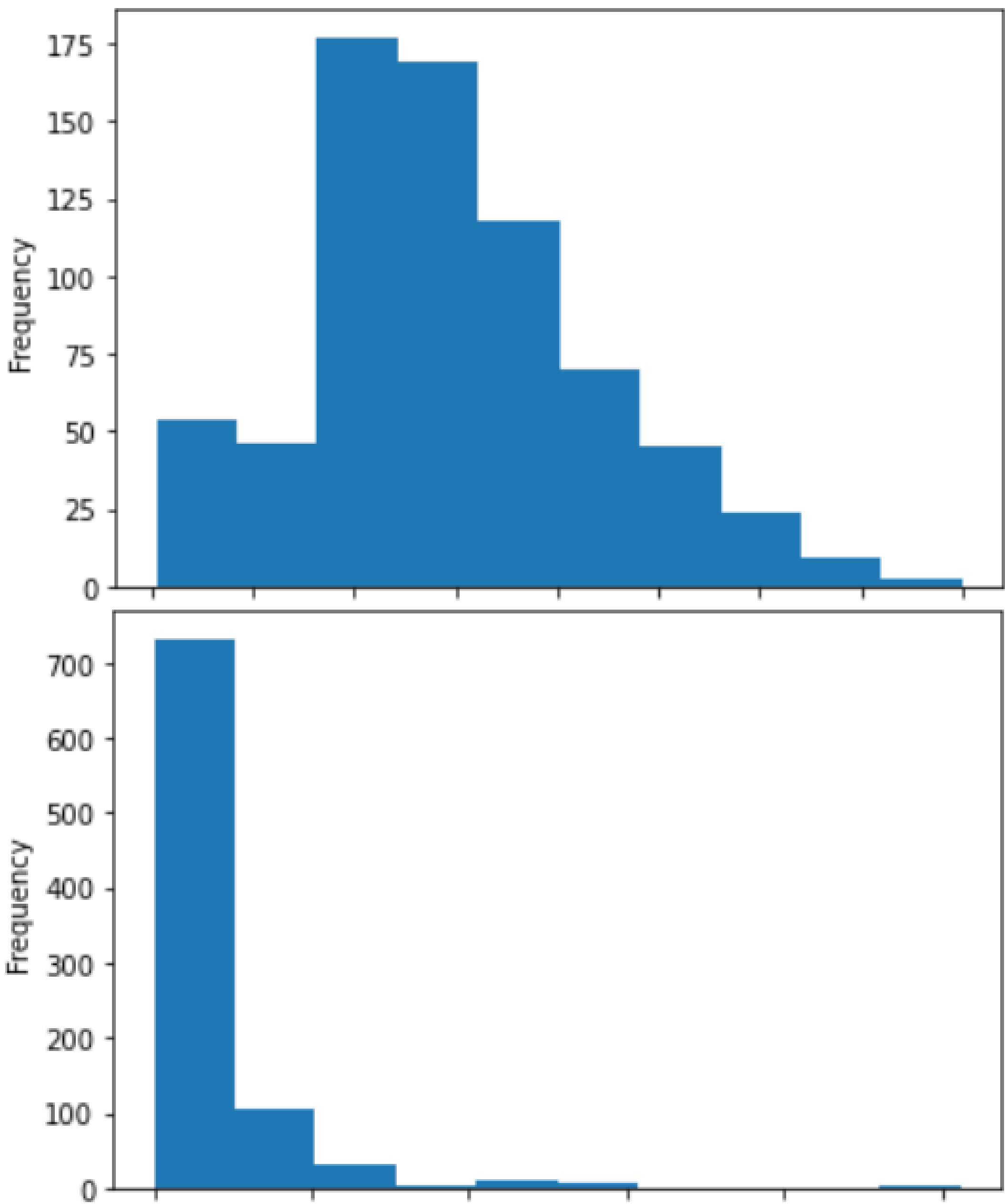
```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.10

Load the data from space_titanic.csv. Examine the data in the dataset and determine which graph corresponds to the correct visualization of the age distribution of passengers on the ship.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```


Frequency
2000
1750
1500
1250
1000
750
500
250
0
Frequency
8000
7000
6000
5000
4000
3000
2000
1000
0
Frequency
8000
7000
6000
5000
4000
3000
2000
1000
0



8.1.11

Load the data from space_titanic.csv. Examine the data in the dataset to see which deck had the most passengers. Also, list the number of passengers accommodated on that deck.

```
A: 142
```

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.1.12

Load the data from space_titanic.csv. Examine the data in the dataset and determine what percentage of passengers were transported to another dimension after encountering the anomaly. Round the result to whole numbers.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2 Spaceship Titanic - working with data

8.2.1

In the first part, we focused on finding out what the dataset regarding the Titanic spacecraft actually looks like and what the characteristics of the passengers who sailed on the ship are. In the second part, we focus on examining those passengers who were transported to another dimension after the ship crash. The aim is to investigate if there is any correlation between some of the characteristics/parameters of the passengers on the ship and their transfer to another dimension.

8.2.2

Load the data from space_titanic.csv. Examine the data in the dataset to see what the percentage distribution of the planet of origin of the passengers *transferred* was. In other words, we want to know what percentage of passengers were transferred to another dimension based on their home planet (note that the sum of these percentages may not add up to 100). Round the result to integers and write it out in the following form:

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.3

Load the data from space_titanic.csv. Examine the data in the dataset to see what happened to the cryo-sleep passengers after the anomaly crash.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
```

```
# explore the dataset
```

- about a third of the passengers sailed in cryo-sleep
- more than two-thirds of the cryo-passengers have been transported to another dimension
- about a third of the passengers who were not in the cryo-ship were transported to another dimension
- more than two-thirds of the passengers sailed in cryo-sleep
- half of the passengers sailed in cryo-sleep
- half of the cryo-passengers have been transported to another dimension
- less than a third of the cryo-passengers have been transported to another dimension
- more than two-thirds of the passengers who were not in the cryo-ship were transported to another dimension
- half of the passengers who weren't in the cryo-ship were transported to another dimension

8.2.4

Load the data from space_titanic.csv. Examine the data in the dataset and determine which deck had the most passengers transferred to another dimension after the crash. Print, along with the name of the deck, the number of passengers transferred.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.5

Load the data from space_titanic.csv. Examine the data in the dataset and determine which side of the deck had the most passengers moved to another dimension after the crash. List, along with the name of the side (Portside or Starboard), the number of passengers moved.

```
Portside: 142
```

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.6

Load the data from space_titanic.csv. Examine the data in the dataset and see what the difference was in passenger spending on luxury ship services. Compare the

average spending of passengers who were transferred to another dimension and those who were not. Round the resulting consumption to two decimal places and print it in the following format:

```
Transported: 153.52 Saved: 133.41
```

Don't forget to fill in the missing values in the examined variables!

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.7

Load the data from space_titanic.csv. Examine the data in the dataset and determine the age group of passengers that were most transported to another dimension. Create the age groups on a 10-year interval. Print the result in the following format, including the number of passengers moved:

```
60-69: 785
```

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.8

Load the data from space_titanic.csv. Examine the data in the dataset and determine the age group of passengers that were most likely to move to another dimension. Create the age groups on a 10-year interval. Report the result in the following format, including the percentage of passengers transferred, rounded to two decimal places:

```
60-69: 55.55
```

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

8.2.9

Load the data from space_titanic.csv. Examine the data in the dataset to see how many passengers who were travelling in groups were moved to another dimension

(there were at least two in the group). The group information can be found in the PassengerID variable.

```
# import library
# read csv from
https://priscilla.fitped.eu/data/pandas/space_titanic.csv
# explore the dataset
```

PRISCILLA

priscilla.fitped.eu