0-Introduction

October 4, 2023

1 Python Crash Course

In this crash course we will see a lot of different concepts in Python. In the following notebooks, we will go through some of the most relevant aspects of Python that we considered you should know. Of course, this will not be an exhaustive introduction, as each task might require some very specific tools. There might be some objects or commands you later find useful but have not seen in this course. However, you really become “fluent” in a programming language by trying to implement your own projects and learn from difficulties you encounter along the way.

Note that we cover **Python 3** here. There exists a lot of code on the Internet (or in your group) written in earlier versions such as Python 2.7. Usually you can identify Python 2.7 code because you find print statements like

```
print "This prints an output in Python 2"
```

which in Python 3 are print functions like

```
print("This prints an output in Python 3")
```

1.0.1 Small overview of working with Jupyter Lab

In this course, we make use of Jupyter Notebooks which allow to combine executable code with text in one document, increasing code-readibility and documentation.

In Jupyter Notebooks, you have different kinds of **cells** you can make use of: **Code**, **Markdown**, and **Raw**. In order to select a cell for execution you can just click on the relevant part of the notebook and a blue bar on the left will highlight the currently selected cell like in the image below.

Try to run the following code cell with in the panel above or *Shift+Enter*!

[2]:
```
print("It works!")
```

```
It works!
```

You can stop the execution of a cell with in the panel!

1.0.2 Markdown mode

Try to add a cell below, change to **Markdown** mode and write a note!

1. Add a cell with + in the panel above or *Esc+B*
2. Change to **Markdown** in the panel above (drop-down list) or *Esc+M*
3. As before, run the cell with in the panel above or *Shift+Enter*!

You can make notes inside the notebooks if you like! In order to edit a Markdown cell double-click somewhere in the cell!

In Markdown mode you can use the following special characters:

2 Title

2.1 Subtitle

- bullet point 1
- bullet point 2

italic font **bold**

`highlighted code`

*** (that's a separator line)

You can also use LaTeX if you want, like $\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$ (double-click here to see how it is used).

2.1.1 Code mode

Creating a **Code** cell works similarly. You can make use of it later on!

1. Add a cell with + in the panel above or *Esc+B*
2. Change to **Code** in the panel above (drop-down list) or *Esc+Y*
3. Again, run the cell with in the panel above or *Shift+Enter*!

2.1.2 Some Remarks:

- Delete a cell with right-click on the cell and the "Delete Cells" option
- **Be careful which cells you delete** because it might not be possible to retrieve them!
- In case, you accidently deleted something try *Esc+Z* to retrieve a cell or *Ctrl+Z* to retrieve changes within a cell
- Save your notebooks with the *save icon* in the panel above or with *Ctrl+S*! **Try to save after every bigger change!** Depending on your machine, Jupyter Lab might crash from time to time.

- Try not to open too many windows in Jupyter Lab. This might lead to a crash sometimes.

2.1.3 The Zen of Python

Run the next cell to learn what kind of philosophy programming in Python is supposed to follow.

[3]:
```python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.1.4 Final Test

Please execute the following cells to check whether you are all set up for the course!

First, you multiply a vector

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

with a scalar value 5, which of course should result in the output

$$\begin{pmatrix} 5 \\ 10 \\ 15 \end{pmatrix}$$

(printed out as `[5 10 15]`).

[4]:
```python
import numpy as np

vector = np.array([1,2,3])
scalar = 5

print(vector*scalar)
```

```
[ 5 10 15]
```

Lastly, let's check whether all libraries are available which we will use later on. If one of these is missing, try to install it!

[5]:
```python
import numpy
import pandas
import matplotlib
import seaborn
import scipy
import sklearn

print("All there!")
```

```
All there!
```

1-Variables_and_Arithmetic

October 4, 2023

1 1. Variables and Arithmetic

In this first section, we learn some basics on variables in Python. In particular, we learn how to * assign and print out variables, * convert them from one type to another, * perform (standard) calculations, and * use logical operators.

Keywords: `type`, `=`, `print`, `int`, `float`, `str`, `bool`, `==`, `<=`, `is`, `and`, `or`

1.1 Initialise and print variables

The two most commonly used number data types are **integers**, i.e. whole numbers, like

[1]:
```
type(1)
```

[1]:
```
int
```

and **floating-point numbers** or **floats**, i.e. decimal numbers, like

[2]:
```
type(2.5)
```

[2]:
```
float
```

Another important data type are strings which are used to represent text

[3]:
```
type("like this text.")
```

[3]:
```
str
```

For assigning values to variables, the `=` operator is used

[4]:
```
integer_variable = 10
float_variable = 2.5
complex_variable = 1 + 2j
string_variable = "some text"
another_string_variable = 'this is also a string'
```

[5]:
```
integer_variable
```

[5]:
```
10
```

[6]:
```
print(integer_variable)
```

```
10
```

Note the difference between using `integer_variable` in the cell above and `print(integer_variable)`. In the first case, Jupyter Notebooks allow us to inspect the internal value of the variable as it would be used in the program (note the `[5]:` indiciating the last output of this particular cell). In the second case, `print` is what you would require if you want to output something during the execution of the program.

[7]:
```
type(integer_variable)
```

[7]:
```
int
```

[8]:
```
print(float_variable)
```

```
2.5
```

[9]:
```
type(float_variable)
```

[9]:
```
float
```

[10]:
```
string_variable
```

[10]:
```
'some text'
```

[11]:
```
print(another_string_variable)
```

```
this is also a string
```

[12]:
```
complex_variable
```

[12]:
```
(1+2j)
```

[13]:
```
a = 127
print(a)
```

```
127
```

[14]:
```
a = 1.1
print(a)
```

```
1.1
```

1.2 Converting variables between different data types

Frequently, we start working with variables which are of a particular type but notice half way through that we actually require another type. Type conversion is not a problem in Python. In fact, we change the type on the fly.

[15]:
```
new_int = 4

print("Initially, we had new_int =", new_int, "of type", type(new_int))

new_int = float(new_int)

print("Now, we have new_int =", new_int, "of type", type(new_int))
```

```
Initially, we had new_int = 4 of type <class 'int'>
Now, we have new_int = 4.0 of type <class 'float'>
```

Note that in this example `<class '...'>` has something to do with the data type. You don't need to pay too much attention to this for the time being.

[16]:
```
int(2.7)
```

[16]: 2

[17]:
```
int('15')
```

[17]: 15

[18]:
```
float('5.7')
```

[18]: 5.7

[19]:
```
converted_int = str(integer_variable)

print("The converted integer is of type", type(converted_int))
converted_int
```

```
The converted integer is of type <class 'str'>
```

[19]: '10'

1.3 Arithmetic operators

Using arithmetic operations (+, -, /, *, //, **, %) should exactly work as you would expect it.

[20]:
```
10 + 2
```

[20]: 12

[21]:
```
integer_variable + float_variable
```

[21]: 12.5

[22]:
```
'example' + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 'example' + 3

TypeError: can only concatenate str (not "int") to str
```

[23]:
```
'example' + str(3)
```

[23]: 'example3'

[24]:
```
100 - 1
```

[24]: 99

[25]:
```
100 - (-1)
```

[25]: 101

[26]:
```
integer_variable - float_variable
```

[26]: 7.5

[27]:
```
100 * -2
```

[27]: -200

[28]:
```
11/2
```

[28]: 5.5

[29]:
```
11//2
```

[29]: 5

[30]:
```
int(11/2)
```

[30]: 5

Note that the **integer division** (//) operation `11//2` and `int(11/2)` both give you the same result which is the integer before the decimal point, i.e. decimals are clipped away.

On the other hand, the **modulus** operator % returns the remainder of a division. For example:

[31]:
```
11%2
```

[31]: 1

[32]:
```
print("2 divides 11 =>", 11//2, "times with a remainder of =>", 11%2)
```

```
2 divides 11 => 5 times with a remainder of => 1
```

[33]:
```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[33], line 1
----> 1 1 / 0

ZeroDivisionError: division by zero
```

For exponentiation in Python we can use the `**` operator or `pow` function. So, for 2^3 we do

[34]:
```
2**3
```

[34]: 8

or

[35]:
```
pow(2,3)
```

[35]: 8

[37]:
```
9**0.5
```

[37]: 3.0

1.4 Logical operators & more

Boolean variables represent another important data type which only allow two values, `True` and `False`.

[38]:
```
bool(1)
```

[38]: True

[39]:
```
bool(0)
```

[39]: False

[40]:
```
bool(100)
```

[40]: True

[41]:
```
bool(0.5)
```

[41]: True

[42]:
```
bool(1) == True
```

[42]: True

[43]:
```
bool(0.5) == False
```

[43]: False

[44]:
```
1 == True
```

[44]: True

[45]:
```
0 == False
```

[45]: True

Note that (in most programming languages) the mathematical symbol for *equals* is redefined as the `==` operator. This is necessary because `=` is already used for assigning values to variables.

[48]:
```
bool_variable = True
placeholder = None
```

[51]:
```
placeholder is None
```

[51]: True

[52]:
```
bool_variable = True

not bool_variable
```

[52]: False

[59]:
```
5 == 5.0
```

[59]: False

[60]:
```
int(5) == float(5)
```

[60]: True

[55]:
```
10 != 10.0
```

[55]: False

[61]:
```
10 <= 10
```

[61]: True

[62]:
```
10 >= 11
```

[62]: False

[63]:
```
"a" > "b"
```

[63]: False

Note that strings are ordered lexicographically.

[64]:
```
'car' < 'cat'
```

[64]: True

Suppose you are looking for new flat and you find an offer located in Basel which has a monthly rent of 500 CHF.

[65]:
```
rent = 500
city = 'Basel'
```

Consider three scenarios: In the first, you want to pay less than 700 CHF and live in Basel. Is this a relevant offer for you?

[66]:
```
rent < 700 and city == 'Basel'
```

[66]: True

In the second scenario, you would like to pay less than 700 CHF or live in Zurich. Would you be interested?

[67]:
```
rent < 700 or city == 'Zurich'
```

[67]: True

In the last scenario, you would like to pay less than 400 CHF or live in Zurich. What about now?

[68]:
```
rent < 400 or city == 'Zurich'
```

[68]: False

1.5 Some caveats

[69]:
```
0.1*3
```

[69]: 0.30000000000000004

This is because the binary representation of 0.1 is a periodically repeating number, i.e.

$$0.1_{\text{decimal}} \mathrel{\hat{=}} 0.0\overline{0011}_{\text{binary}}.$$

The representation of this number needs to be clipped at some point, as we have a limited memory resources to represent numbers.

[70]:
```
5/11
```

[70]:
```
0.45454545454545453
```

[71]:
```
2e400
```

[71]:
```
inf
```

You have a lot of freedom in deciding the name of your variable. You can use underscores (*_*), numbers, or capital letters like

[72]:
```
integer2 = 15
integer_variable = 10
integerVariable = 5
```

However, the following would not work, as they are **illegal variable names**:

[73]:
```
2integer = 15
```

```
<>:1: SyntaxWarning: invalid decimal literal
<>:1: SyntaxWarning: invalid decimal literal
/var/folders/y_/2qg_gjq937z8c0xctbn85x8r0000gn/T/ipykernel_1945/2442139553.py:1:
SyntaxWarning: invalid decimal literal
  2integer = 15
```

```
  Cell In[73], line 1
    2integer = 15
     ^
SyntaxError: invalid syntax
```

[74]:
```
integer-variable = 10
```

```
  Cell In[74], line 1
    integer-variable = 10
    ^
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of
 ↪'='?
```

[75]:
```
integer variable = 5
```

```
  Cell In[75], line 1
    integer variable = 5
```

```
              ^
SyntaxError: invalid syntax
```

1.6 Exercise section

(1.) Assign to two different variables the values `123` and `123.0`.

[78]:
```
a = 123
b = 123.0
```

(2.) Check with `==` whether these two variables are equal and save the result to a new variable.

[83]:
```
c = a == b
```

(3.) Print the following sentences using strings, the two variables from (1.) and the boolean (result) variable from (2.) to output the following sentences:

`Comparing the two variables leads to the result: 123 == 123.0 is True`

[84]:
```
print('Comparing ... results:', a, '==', b, 'is', c)
```

```
Comparing … results: 123 == 123.0 is True
```

1.7 Proposed Solution

(1.) Assign to two different variables the values `123` and `123.0`.

[1]:
```
my_int = 123
my_float = 123.0
```

(2.) Check with `==` whether these two variables are equal and save the result to a new variable.

[2]:
```
my_result = 123 == 123.0
```

(3.) Print the following sentences using strings, the two variables from (1.) and the boolean (result) variable from (2.) to output the following sentences:

`Comparing the two variables leads to the result: 123 == 123.0 is True`

[3]:
```
print('Comparing the two variables leads to the result:', my_int, '==',␣
↪my_float, 'is', my_result)
```

```
Comparing the two variables leads to the result: 123 == 123.0 is True
```

2-Data_Structures

October 4, 2023

1 2. Data Structures

In the second section we learn about data structures such as lists, dictionaries, tuples, and sets. This includes

- when to use which structure,
- how to change the elements of the data structures,
- how indexing in Python works,
- nested structures (e.g. a list in a list) and
- a little bit more on strings.

Keywords: `list`, `dict`, `len`, `append`, `extend`, `insert`, `remove`, `del`, `help`, `pop`, `sort`, `::2`, `split`, `items`, `keys`, `values`, `tuples`, `add`, `discard`

1.1 Lists

A list is a sequence of ordered values with each element being indexed by an integer.

[1]:
```
days = ['Monday','Tuesday','Wednesday']
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday']
```

[2]:
```
days[0]
```

[2]:
```
'Monday'
```

Note that the indexing in Python starts with 0.

[3]:
```
type(days)
```

[3]:
```
list
```

[4]:
```
len(days)
```

[4]:
```
3
```

Note that with `len` the length of the list is obtained.

[5]:
```
days[1]
```

[5]: 'Tuesday'

[6]:
```
days[2]
```

[6]: 'Wednesday'

[7]:
```
days[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 days[3]

IndexError: list index out of range
```

Lists are mutable In the following we will apply list **methods** which allow to perform manipulation of the list objects. You identify a list method by the list name followed by a dot and the name of the list method.

Here we see how to change the content of a list. All the changes happen *in place*, i.e. you don't need to assign the resulting changes to the list.

[8]:
```
days.append('Friday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Friday']
```

[9]:
```
days.extend( ['Saturday', 'Sunday'] )
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Friday', 'Saturday', 'Sunday']
```

[10]:
```
days.insert(3, 'Thursday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

[11]:
```
days.pop()
```

[11]: 'Sunday'

[12]:
```
days
```

[12]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

[13]:
```
days.insert(0, 'Friday')
print(days)
```

```
['Friday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

[14]:
```
days.remove('Friday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

[15]:
```
days
```

[15]:
```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

Note that the `remove` method only removes one occurence of the specified element. Let's perform the removal again:

[16]:
```
days.remove('Friday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Saturday']
```

[17]:
```
del days[0]
print(days)
```

```
['Tuesday', 'Wednesday', 'Thursday', 'Saturday']
```

Note that you can also delete by index with `del`.

[18]:
```
help(list.pop)
```

```
Help on method_descriptor:

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

Note that you can use `help` to display some documentation and learn e.g. a bit more about particular methods for lists.

Let's add the missing days again:

[19]:
```
days.insert(0,'Monday')
days.insert(4,'Friday')
days.append('Sunday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

You can initialise an empty list with `list()` or `[]`.

[20]:
```
numbers = list()
print("The initialised list is empty:", numbers)
numbers.extend( [14,2,5,1,101] )
print("Now we filled it with numbers:", numbers)
```

```
The initialised list is empty: []
Now we filled it with numbers: [14, 2, 5, 1, 101]
```

[21]:
```
numbers.sort()
print(numbers)

numbers.sort(reverse=True)
print(numbers)
```

```
[1, 2, 5, 14, 101]
[101, 14, 5, 2, 1]
```

[22]:
```
numbers + days
```

[22]:
```
[101,
 14,
 5,
 2,
 1,
 'Monday',
 'Tuesday',
 'Wednesday',
 'Thursday',
 'Friday',
 'Saturday',
 'Sunday']
```

Note that you would get the same result with

`numbers.extend(days)`

[23]:
```
numbers*2
```

[23]:
```
[101, 14, 5, 2, 1, 101, 14, 5, 2, 1]
```

Note that you would get the same result with

`numbers.extend(numbers)`

1.1.1 More on indexing

[24]:
```
days
```

[24]:
```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

[28]:
```
days[1:3]
```

[28]:
```
['Tuesday', 'Wednesday']
```

[29]:
```
days[:5]
```

[29]:
```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

[30]:
```
weekend = days[5:]
print(weekend)
```

```
['Saturday', 'Sunday']
```

[32]:
```
days[::2]
```

[32]:
```
['Monday', 'Thursday', 'Sunday']
```

Note that you can also use negative indices to query list from the end.

[35]:
```
days[-1]
```

[35]:
```
'Sunday'
```

1.2 Nested lists

A list can contain lists itself. You can for example think of a matrix as a list of lists as in the following example.

[36]:
```
matrix = [ [1,2] , [3,4] ]
```

[37]:
```
matrix
```

[37]:
```
[[1, 2], [3, 4]]
```

[38]:
```
matrix[0]
```

[38]:
```
[1, 2]
```

[39]:
```
matrix[0][0]
```

[39]:
```
1
```

[40]:
```
matrix[1][1]
```

[40]:
```
4
```

[41]:
```
matrix[1][1] = 400
print(matrix)
```

```
[[1, 2], [3, 400]]
```

Note that nested lists can be as deep as you need them. For example, you can construct a 3 dimensional matrix (a tensor) with nested lists.

[42]:
```
tensor = [ [[1,2],[3,4]] ,[[5,6],[7,8]] , [[9,10],[11,12]] ]
tensor
```

[42]:
```
[[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]]
```

[43]:
```
new_list = ['a', 3, 1.5, [1,2,3]]
```

[44]:
```
print(new_list)
```

```
['a', 3, 1.5]
```

1.3 A little bit more on strings

You can think of a string as a list where each letter has an index. For

[45]:
```
string = "This is an example"
```

the indices would be:

T	h	i	s	_	i	s	_	a	n	_	e	x	a	m	p	l	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

[46]:
```
string[0]
```

[46]:
```
'T'
```

[47]:
```
string[:7]
```

[47]:
```
'This is'
```

[48]:
```
string[4]
```

[48]:
```
' '
```

[49]:
```
new_string = string + " we would like to extend"
new_string
```

[49]:
```
'This is an example we would like to extend'
```

[55]:
```
new_string.split(" ")
```

[55]:
```
['This', 'is', 'an', 'example', 'we', 'would', 'like', 'to', 'extend']
```

[51]:
```
string[0] = 't'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 string[0] = 't'

TypeError: 'str' object does not support item assignment
```

[52]:
```
integer_string = '123'
integer_string*3
```

[52]: '123123123'

[53]:
```
len(integer_string)
```

[53]: 3

1.4 Dictionaries

A special data structure in Python is the dictionary, where **key-value** pairs can be defined. Suppose you would like to encode a table with the country calling prefix codes like

Country	Code
France	33
Germany	49
Italy	39
Switzerland	41
UK	44

[56]:
```
country_codes = {'Switzerland': 41, 'France': 33, 'Italy': 39,
                 'UK': 44, 'Germany': 49}
country_codes
```

[56]: {'Switzerland': 41, 'France': 33, 'Italy': 39, 'UK': 44, 'Germany': 49}

[57]:
```
country_codes['Italy']
```

[57]: 39

Dictionaries have special methods to query the contained information:

[58]:
```
country_codes.items()
```

[58]: dict_items([('Switzerland', 41), ('France', 33), ('Italy', 39), ('UK', 44), ('Germany', 49)])

[59]:
```
country_codes.values()
```

[59]: dict_values([41, 33, 39, 44, 49])

[60]:
```
country_codes.keys()
```

[60]: dict_keys(['Switzerland', 'France', 'Italy', 'UK', 'Germany'])

Adding new entries to the dictionary is straight forward:

[61]:
```
country_codes['Spain'] = 34
print(country_codes)
```

```
{'Switzerland': 41, 'France': 33, 'Italy': 39, 'UK': 44, 'Germany': 49, 'Spain':
34}
```

[62]:
```
del country_codes['UK']
country_codes
```

[62]: {'Switzerland': 41, 'France': 33, 'Italy': 39, 'Germany': 49, 'Spain': 34}

[63]:
```
country_codes['UK']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[63], line 1
----> 1 country_codes['UK']

KeyError: 'UK'
```

[64]:
```
country_codes.get('Italy')
```

[64]: 39

[67]:
```
country_codes.get('UK')
```

[68]:
```
new_var = country_codes.get('UK')
print(new_var)
```

```
None
```

[69]:
```
country_codes
```

[69]: {'Switzerland': 41, 'France': 33, 'Italy': 39, 'Germany': 49, 'Spain': 34}

[70]:
```
dict_pop = country_codes.pop("Italy", None)
```

[71]:
```
print(dict_pop)
```

```
39
```

[72]:
```
country_codes
```

[72]:
```
{'Switzerland': 41, 'France': 33, 'Germany': 49, 'Spain': 34}
```

[74]:
```
dict_pop = country_codes.pop("UK", 999)
print(dict_pop)
```

```
999
```

Note that `get` and `pop` are useful methods to access potential elements of a dictionary, which do not issue an error when an element is not in the dictionary.

[75]:
```
nested_dict = {'Switzerland': {'Capital': 'Bern', 'Country_Code': 41},
               'France': {'Capital': 'Paris', 'Country_Code': 33},
               'Italy': {'Capital': 'Rome', 'Country_Code': 39},
               'Germany': {'Capital': 'Berlin', 'Country_Code': 49}
              }
```

[76]:
```
nested_dict['France']
```

[76]:
```
{'Capital': 'Paris', 'Country_Code': 33}
```

[77]:
```
nested_dict['France']['Capital']
```

[77]:
```
'Paris'
```

Checking if a key is present in the dictionary:

[78]:
```
'France' in nested_dict
```

[78]:
```
True
```

[79]:
```
'UK' in nested_dict
```

[79]:
```
False
```

[80]:
```
country_codes
```

[80]:
```
{'Switzerland': 41, 'France': 33, 'Germany': 49, 'Spain': 34}
```

[81]:
```
other_dict = {1: 'a', 2: 'b'}
other_dict
```

[81]:
```
{1: 'a', 2: 'b'}
```

1.5 Other data structures: Tuples & sets

[83]:
```
my_tuple = (1,2,3,4)
```

[84]:
```
len(my_tuple)
```

[84]: 4

[85]:
```
my_tuple[1]
```

[85]: 2

Note that tuples are immutable, you can't change the contained elements.

[86]:
```
my_tuple[1] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[86], line 1
----> 1 my_tuple[1] = 100

TypeError: 'tuple' object does not support item assignment
```

[91]:
```
my_set = {'a','b','c','d','c','b','a'}
print(my_set)
```

```
{'d', 'b', 'a', 'c'}
```

[92]:
```
my_set.add('e')
print(my_set)
```

```
{'a', 'c', 'd', 'e', 'b'}
```

[93]:
```
my_set.discard('a')
print(my_set)
```

```
{'c', 'd', 'e', 'b'}
```

[94]:
```
list(my_set)
```

[94]: ['c', 'd', 'e', 'b']

1.6 Some caveats

[1]:
```
matrix = [ [1,2],[3,4] ]

copy_matrix = matrix
print(copy_matrix)
```

```
[[1, 2], [3, 4]]
```

[2]:
```
copy_matrix[1][1] = 40
print("The copied matrix looks like\n", copy_matrix)
print("But also the original matrix looks now like\n", matrix)
```

```
The copied matrix looks like
 [[1, 2], [3, 40]]
But also the original matrix looks now like
 [[1, 2], [3, 40]]
```

What is the problem here? It is because

[98]:
```
copy_matrix is matrix
```

[98]:
```
True
```

How to really copy:

[4]:
```
import copy

matrix = [[1,2], [3,4]]
real_copy = copy.deepcopy(matrix)

print("Real copy before:", real_copy)
print("Matrix before:", matrix)

print("Assign 40")
real_copy[1][1] = 40

print("Real copy after:", real_copy)
print("Matrix after:", matrix)
```

```
Real copy before: [[1, 2], [3, 4]]
Matrix before: [[1, 2], [3, 4]]
Assign 40
Real copy after: [[1, 2], [3, 40]]
Matrix after: [[1, 2], [3, 4]]
```

Note that you should not use variable names which are already used by Python in some way. For example, do not use built-in function names as variable names.

So something like the following would be a bad idea

```
list = [1, 10, 100]
dict = {'a':1, 'b':2}
print = "Just a string"
```

1.7 Exercise section

(1.) Rearange the following list so that the resulting list displays all integers from 1 to 8, i.e. `[1, 2, 3, 4, 5, 6, 7, 8]`.

[1]:
```
new_numbers = [9,7,6,2]
```

Put your solution here:

[ ]:
```

```

[ ]:
```
print("Your solution is", new_numbers)
```

(2.) Add to dictionary

[1]:
```
constants = {'c': 299792458}
```

the following additional constants

- $\pi = 3.14159$
- $e = 2.71828$
- $\phi = 1.61803$

where you use *pi, e, phi* as the keywords and the respective floats as the values.

Put your solution here:

[ ]:
```

```

[ ]:
```
print(constants)
```

1.8 Proposed Solution

(1.) Rearange the following list so that the resulting list displays all integers from 1 to 8:

[1]:
```
new_numbers = [9,7,6,2]
```

Put your solution here:

[10]:
```
new_numbers.extend([1,3,4,5,8])
new_numbers.sort()
new_numbers.remove(9)
```

[11]:
```
print("Your solution is", new_numbers)
```

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

(2.) Add to dictionary

[2]:
```
constants = {'c': 299792458}
```

the following additional constants

- $\pi = 3.14159$
- $e = 2.71828$
- $\phi = 1.61803$

where you use *pi*, *e*, *phi* as the keywords and the respective floats as the values.

Put your solution here:

[3]:
```
constants['pi'] = 3.14159
constants['e'] = 2.71828
constants['phi'] = 1.61803
```

[4]:
```
print(constants)
```

```
{'c': 299792458, 'pi': 3.14159, 'e': 2.71828, 'phi': 1.61803}
```

3-Conditional_Statements

October 4, 2023

1 3. Conditional Statements

In the third section we learn how

- to use conditions to cover different cases,
- to place comments in our code,
- indentations structure blocks of grouped statements,
- to work with error messages and
- to avoid our code stopping when encountering an error with exceptions.

Keywords: `if`, `else`, `elif`, `try`, `except`, `assert`, `# comment`, `'''comment'''`

1.1 if-Statement

Conditional statements start with the `if` keyword and a **condition** which needs to be fulfilled in order to execute the code in the **body**. Note that you have to set a colon `:` after the condition statement.

[6]:
```
threshold_value = 0.3
```

[4]:
```
if threshold_value >= 0.5:
    print("The threshold value exceeds 0.5!")
```

```
The threshold value exceeds 0.5!
```

[7]:
```
if threshold_value >= 0.5:
    # This is a comment which has no effect on the
    # code. You can use comments to explain what
    # your code is doing. This is usually advisable
    # whenever you write a script!
    # In this case, threshold_value exceeds 0.5.

    print("The threshold_value exceeds 0.5!")

else:
    ''' This is comment which spans multiple lines
    without the necessity of setting each time a #
    in front.
```

```
    In this case, threshold_value is less than 0.5.
    '''

    print("Threshold was not met!")

print("-> This comes afterwards!")
```

```
Threshold was not met!
-> This comes afterwards!
```

[9]:
```
new_value = 0.3
case = 'B'

if new_value >= 0.5:
    print("The threshold value exceeds 0.5!")

    if case == 'A':
        print("We are in case A.")
    else:
        print("We are in another case.")
else:
    print("Threshold was not met!")

    if case == 'B':
        print("We are in case B.")
    else:
        print("We are in another case.")

print("!")
```

```
Threshold was not met!
We are in case B.
!
```

Note that blocks of code which belong together are **indented in exactly the same way**. In Python, brackets like `{ }` are not required to group statements. Usually, indentation is a recommend way of making a code easier to read. Python makes this recommendation to some extent mandatory. Make sure to use the same indentation style throughout your code, either **4 spaces** or **1 tab** per level. Otherwise, running your code might lead to errors.

In many other programming languages this is different. In R for example this might look something like this:

```
if (threshold_value >= 0.5) {
print("The threshold value exceeds 0.5!")
} else {
print("Threshold was not met!")
}
```

[12]:
```
name = 'Wonty'
name_starts_with = name[0]
name_starts_with
```

[12]: 'W'

[13]:
```
if name_starts_with in ['A','B','C','D','E','F']:
    print("Go to room 1.")

elif name_starts_with in ['G','H','I','J','K','L']:
    print("Go to room 2.")

elif name_starts_with in ['M','N','O','P','Q','R']:
    print("Go to room 3.")

else:
    print("Go to room 4.")
```

```
Go to room 4.
```

[14]:
```
str_variable = 'T'
float_variable = 12.0
my_list = [1,2,3]
empty = None
```

[15]:
```
if str_variable:
    print("This worked.")
```

```
This worked.
```

[19]:
```
bool(str_variable)
```

[19]: True

[20]:
```
if float_variable:
    print("This worked, too.")
```

```
This worked, too.
```

[21]:
```
if my_list:
    print("This worked, wow!")
```

```
This worked, wow!
```

[26]:
```
another_int = 0

print(bool(another_int))

if another_int:
    print("?")
```

```
else:
    print("!")
```

```
False
!
```

[23]:
```
if empty:
    print("Does this work?")
```

[27]:
```
grade = 3.5

passed = True if grade >= 4.0 else False

print("Did you pass? Answer:", passed)
```

```
Did you pass? Answer: False
```

Note that

```
passed = True if grade >= 4.0 else False
```

is the one-line version of

```
if grade >= 4.0:
    passed = True
else:
    passed = False
```

Note that this example is intended to illustrate the one-line if-else statement. The more direct way with the same result would be

```
passed = grade >= 4.0
```

1.2 Error messages and handling exceptions

Sometimes your code doesn't run because an error occured. Often the error message tells you exactly what went wrong. In case, you don't really understand the error message, the standard way to identify the problem consists by searching for the last line of the error message online and see if anyone encountered the same problem earlier. :)

A way of addressing problems which are known to occur is by using exceptions with the `try` and `except` keywords. In this way, your programm doesn't need to stop and can be further executed.

[28]:
```
if we_didnt_define_this:
    print("No.")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[28], line 1
```

```
----> 1 if we_didnt_define_this:
      2     print("No.")

NameError: name 'we_didnt_define_this' is not defined
```

[29]:
```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[29], line 1
----> 1 1 / 0

ZeroDivisionError: division by zero
```

[30]:
```
pow(10.0,1000000000000000)
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
Cell In[30], line 1
----> 1 pow(10.0,1000000000000000)

OverflowError: (34, 'Result too large')
```

[31]:
```
"string" + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 1
----> 1 "string" + 1

TypeError: can only concatenate str (not "int") to str
```

Note that you can use the types of error like `TypeError`, `OverflowError`, `ZeroDivisionError` and `NameError` to define your own exceptions as in the following.

[32]:
```
try:
    print(1 / 0)
    # print("string" + 1)

except ZeroDivisionError:
    print("Divison with 0")
```

```
Divison with 0
```

[33]:
```
try:
    print("string" + 1)

except TypeError:
    print("string1")
```

```
string1
```

[34]:
```
try:
    print(some_variable + 1)

except:
    print("Define the variable first!")
```

```
Define the variable first!
```

[41]:
```
positive_number = -1
```

[42]:
```
print("Is positive_number larger 0? ->", positive_number > 0)
```

```
Is positive_number larger 0? -> False
```

[43]:
```
assert positive_number > 0

print("Let's continue with our code!")
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[43], line 1
----> 1 assert positive_number > 0
      3 print("Let's continue with our code!")

AssertionError:
```

Note with `assert` you can check whether a condition is fulfilled and only continue with your code if this is the case

`assert condition, "error statement to print if condition is not fulfilled"`

[44]:
```
assert positive_number > 0, "postive_number is actually negative or 0!"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[44], line 1
----> 1 assert positive_number > 0, "postive_number is actually negative or 0!"

AssertionError: postive_number is actually negative or 0!
```

1.3 Some caveats

It is easy to define conditions which show a behaviour other than expected. Consider the following examples where in both cases we would like to have the output `Yes!`

[49]:
```
result = 1/3

if result == 0.3333333333333333:
    print("Yes!")
else:
    print("No!")
```

```
No!
```

[46]:
```
print(result)
```

```
0.3333333333333333
```

[50]:
```
result = 0.1*3

if result == 0.3:
    print("Yes!")
else:
    print("No!")
```

```
No!
```

[51]:
```
print(result)
```

```
0.30000000000000004
```

Recall the modulus operator

[52]:
```
20%2
```

[52]: `0`

[53]:
```
not 20%2
```

[53]: `True`

[54]:
```
not False
```

[54]: `True`

[55]:
```
0 == True
```

[55]: `False`

[56]:
```
a = 20
```

[57]:
```
if not a%2:
    print("The variable can be divided by 2!")
elif not a%4:
    print("The variable can also be divided by 4!")
else:
    print("The variable cannot be divided by either 2 or 4!")
```

```
The variable can be divided by 2!
```

[58]:
```
if not a%2:
    print("The variable can be divided by 2!")

if not a%4:
    print("The variable can also be divided by 4!")

if a%2 and a%4:
    print("The variable cannot be divided by either 2 or 4!")
```

```
The variable can be divided by 2!
The variable can also be divided by 4!
```

[ ]:
```
20 == True
```

[ ]:
```
2 == 1
```

[ ]:
```
bool(2) is True
```

[ ]:
```
1 == True
```

1.4 Exercise section

(1.) What is the problem in the following conditional statement? Try to fix it in any way.

[77]:
```
new_example = 'house'

if new_example[0] >= 's':
    # This is just a comment.
else:
    print("Word starts with a letter before s.")
```

```
Word starts with a letter before s.
```

(2.) What is wrong in the next example? Try to fix it.

[72]:
```
new_bool = True
```

```
if new_bool == 1
 print("Correct")
else
 print("Wrong")
```

```
  Cell In[72], line 3
    if new_bool == 1
                    ^
SyntaxError: expected ':'
```

1.5 Proposed Solutions

(1.) What is the problem in the following conditional statement? Try to fix it in any way.

```
[ ]: new_example = 'house'

if new_example[0] >= 's':
    # This is just a comment.
    print("Word starts with a letter after r")
else:
    print("Word starts with a letter before s.")
```

(2.) What is wrong in the next example? Try to fix it.

```
[ ]: new_bool = True

if new_bool == 1:
    print("Correct")
else:
    print("Wrong")
```

4-Iterations

October 4, 2023

1 4. Iterations

In the fourth section we have a look at performing iterative tasks. We learn how to

- define `for` and `while`-loops,
- use nested loops,
- create list comprehension,
- make use of special statements and functions for loops,
- work a bit differently with strings (again) and
- do debugging with `print`, `type`, `len`.

Keywords: `for`, `while`, `range`, `enumerate`, `zip`, `continue`, `break`, `pass`, `format`

1.1 `for`-Loop

A for loop starts with the `for` keyword followed a membership expression which specifies the elements of the iteration in order to execute the code in the body. Note that you have to set a colon `:` after the membership expression (again). The indentation defines once again which parts of the code belong together and are executed in the loop.

The object over whose members the loop iterates are called **iterables**. In other words, an iterable is something you can iterate / loop over.

[1]:
```
numbers = [0,1,2,3,4,5,6]

for i in numbers:
    # This is the loop body which will be executed iteratively.
    # numbers is an iterable.
    print(i)
    print("End of this block!")
```

```
0
End of this block!
1
End of this block!
2
End of this block!
3
End of this block!
```

```
4
End of this block!
5
End of this block!
6
End of this block!
```

Note that you can use the `range` function to achieve the same result as in the previous loop. `range(7)` gives an object which provides 7 integers from 0 to 6.

[2]:
```
for A in range(7):
    # range gives integers 0 to 6, i.e.
    # it doese not include 7.
    print(A)
```

```
0
1
2
3
4
5
6
```

[3]:
```
num = 100

for num in range(7):
    # num = 0
    # num = 1
    # Perform a calculation
    res = num**2
    print(res)

print("After the loop 'res' is", res)
```

```
0
1
4
9
16
25
36
After the loop 'res' is 36
```

[4]:
```
for letter in 'string':
    print(letter)
```

```
s
t
r
```

```
i
n
g
```

[5]:
```
country_codes = {'Switzerland': 41, 'France': 33, 'Italy': 39,
                 'UK': 44, 'Germany': 49}

for k in country_codes:
    # This block belongs to the loop.
    print(k)

# This part is executed after the loop.
print("\nThis just printed the keys of the dictionary.")
```

```
Switzerland
France
Italy
UK
Germany

This just printed the keys of the dictionary.
```

[6]:
```
print(country_codes.items())
```

```
dict_items([('Switzerland', 41), ('France', 33), ('Italy', 39), ('UK', 44),
('Germany', 49)])
```

[8]:
```
for a in country_codes.items():
    print(a)
    k = a[0]
    v = a[1]
    print(k,v)
    print("---")
```

```
('Switzerland', 41)
Switzerland 41
---
('France', 33)
France 33
---
('Italy', 39)
Italy 39
---
('UK', 44)
UK 44
---
('Germany', 49)
Germany 49
---
```

[10]:
```
for k,v in country_codes.items():
    print(k,v)

print("\nThis printed keys and values of the dictionary.")
```

```
Switzerland 41
France 33
Italy 39
UK 44
Germany 49

This printed keys and values of the dictionary.
```

[14]:
```
for v in country_codes.values():
    print(v)

for k in country_codes.keys():
    print(k)
```

```
41
33
39
44
49
Switzerland
France
Italy
UK
Germany
```

The two functions `enumerate` and `zip` provide some useful functionality for loops. The two functions take iterables as arguments. Let's investigate what they do below.

[19]:
```
colours = ['green', 'red', 'blue', 'yellow']

for col in colours:
    print(col)

print("\nBelow with enumeration!\n")

for index_of_enum, col in enumerate(colours):
    print(index_of_enum, col)

print("\nNew Example:\n")

for e in enumerate(colours):
    print(e)
```

```
green
red
blue
yellow

Below with enumeration!

0 green
1 red
2 blue
3 yellow

New Example:

(0, 'green')
(1, 'red')
(2, 'blue')
(3, 'yellow')
```

[24]:
```
constants = [2.71, 3.14, 1.61]
names = ['e','pi','phi']

for n, c in zip(names, constants):
    print(n,c)
```

```
e 2.71
pi 3.14
phi 1.61
```

[21]:
```
my_dict = dict()

for n, c in zip(names, constants):
    my_dict[n] = c

print(my_dict)
```

```
{'e': 2.71, 'pi': 3.14, 'phi': 1.61}
```

[22]:
```
constants = [2.71, 3.14, 1.61]
names = ['e','pi','phi']
colours = ['green', 'red', 'blue', 'yellow']

for n, c, col in zip(names, constants, colours):
    print(n,c,col)
```

```
e 2.71 green
pi 3.14 red
phi 1.61 blue
```

[23]:
```
const_dict = {'e':2.71, 'pi':3.14, 'phi':1.61}

for k,v in const_dict.items():
    print(k,v)
```

```
e 2.71
pi 3.14
phi 1.61
```

1.1.1 Nested loops

[26]:
```
for i in ['a','b','c']:
    print("----")

    for j in range(6,8):
        print("i =", i, "and j =", j)

print("Continue")
```

```
----
i = 6 and j = a
i = 7 and j = a
----
i = 6 and j = b
i = 7 and j = b
----
i = 6 and j = c
i = 7 and j = c
Continue
```

[ ]:
```
help(range)
```

1.2 While-Loop

A while loop starts with the `while` keyword and a conditional statement, ending with a colon `:`. If the condition is fulfilled, the loop body is executed until the condition is no longer fulfilled. As before, the indentation signals what belongs to the loop and what not.

[27]:
```
counter = 0

while counter < 10:
    counter = counter + 1

    print(counter)

print("\n", counter, "is the final value of counter.")
```

```
1
2
3
4
5
6
7
8
9
10

 10 is the final value of counter.
```

The `while` loop is well-suited for task which **depend on a condition**, whereas the `for` loop is used when a **certain number of repetitions** is required or all elements of an object need to be considered.

1.3 List comprehension

A list comprehension is a concise way to perform a loop and store the result in a list. The following loop

[28]:
```python
result_list = []

for i in range(10):
    result_list.append(i**2)

print(result_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

can be also realised with

[29]:
```python
result_list = [ i**2 for i in range(10) ]

print(result_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can even use conditions, e.g. that `i` should be larger than 3 for storing the result in the list

[30]:
```python
[i**2 for i in range(10) if i > 3]
```

[30]:
```
[16, 25, 36, 49, 64, 81]
```

[32]:
```python
[i**2 for i in range(4,10)]
```

[32]:
```
[16, 25, 36, 49, 64, 81]
```

Some special statements you might find useful when working with loops. This works both with `for` and `while` loops.

[35]:
```
for i in range(10):
    if i%2:
        continue
        print("Something")
    print(i)
```

```
0
2
4
6
8
```

[33]:
```
print(0%2, 1%2, 2%2, 3%2)
```

```
0 1 0 1
```

With `continue` the loop is stopped in the execution of the loop body at that point and the loop continues with the next element of the loop.

[36]:
```
for i in range(10):
    if i%2:
        break
    print(i)

print("We are here now.")
```

```
0
We are here now.
```

With `break` the loop is stopped completely. All other elements of the loop which should follow are ignored.

[37]:
```
for i in range(10):
    if i%2:
        pass
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

With `pass` the loop is executed as usual. When this statement is executed nothing happens. `pass` is often useful as a placeholder. Suppose you want to incorporate some functionality for the case `i%2`, but you didn't have the time yet. Just set `pass` in the body such that your code can be executed without a problem.

1.4 Even more on strings

So far, when combining strings with other variables of any type (`int`, `float`, `str`), we used something like

[38]:
```
"Our result deviates by " + str(0.5) + " percent."
```

[38]:
```
'Our result deviates by 0.5 percent.'
```

or

[39]:
```
print("The integers", 3, "and", 7, "are prime numbers.")
```

```
The integers 3 and 7 are prime numbers.
```

We have seen a similar example for the nested loop. You can achieve the same output as before with the following `print` statement. It makes use of the `format` method for strings and replaces each `{}` in the string with the specified variable accompanied by format (in the same order).

This allows to adjust the formatting a little bit more convenient.

[40]:
```
for i in ['a','b','c']:
    print("----")

    for j in range(6,8):
        print("i = {} and j = {}.".format(i,j))

        # Before:
        # print("i =", i, "and j =", j)
```

```
----
i = a and j = 6.
i = a and j = 7.
----
i = b and j = 6.
i = b and j = 7.
----
i = c and j = 6.
i = c and j = 7.
```

Note that this works for any string!

[41]:
```
experiment_counter = 4
date = '05-11-19'
```

```
'{}_Analysis_{}.csv'.format(date, experiment_counter)
```

[41]: '05-11-19_Analysis_4.csv'

Since Python 3.6 you can also use *f-strings* like this

[42]:
```
f'{date}_Analysis_{experiment_counter}.csv'
```

[42]: '05-11-19_Analysis_4.csv'

[43]:
```
for i in ['a','b','c']:
    for j in range(6,8):
        print(f"i = {i} and j = {j}.")
```

```
i = a and j = 6.
i = a and j = 7.
i = b and j = 6.
i = b and j = 7.
i = c and j = 6.
i = c and j = 7.
```

1.5 Debugging *light*

It is easy to incorporate some unintended behaviour into your program, espacially when using loops. There are *debugging* libraries and software which goes through your code (line by line) and is capable of identifying problems with variable assignment, spelling mistakes, missed syntactical elements like brackets or colons etc. without the necessity to run your code every time completely from the beginning.

For some mistakes, you can rely on the issued error messages. However, a very common and quick way to do debugging is to use the `print`, `type` and `len` functions. The underlying idea is to test whether, some of the intermediate steps have the right size or length, are of the correct type or directly have a look at a variable to identify possible problems.

[44]:
```
initial_list = []

for i in range(8):
    calc_1 = pow(i,2)                # calc_1 = i**2
    initial_list = calc_1

new_variable = initial_list[2]

other_variable = new_variable + 200
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[44], line 7
```

```
      4     calc_1 = pow(i,2)                  # calc_1 = i**2
      5     initial_list = calc_1
----> 7 new_variable = initial_list[2]
      9 other_variable = new_variable + 200

TypeError: 'int' object is not subscriptable
```

[45]:
```
initial_list = []

for i in range(8):
    calc_1 = pow(i,2)
    initial_list = calc_1

print(type(initial_list))

print(initial_list)

# Comment out the problematic line
# new_variable = initial_list[2]

# other_variable = new_variable + 200
```

```
<class 'int'>
49
```

Here goes the correction:

[57]:
```
initial_list = []

for i in range(8):
    calc_1 = pow(i,2)
    initial_list.append(calc_1)
    # initial_list = calc_1

assert type(initial_list) == list, "This is not a list!"

new_variable = initial_list[2]

other_variable = new_variable + 200
```

[58]:
```
print(new_variable, "and" , other_variable)
```

```
4 and 204
```

However, this way of debugging can be very inefficient as you need to execute your code several times and need to know exactly what you want to print or test. A more sophisticated way is to use an interactive source code debugger like pdb, the built-in breakpoint() function (Python version >=3.7) or the function embed() to open an IPython shell within your script.

1.6 Some caveats

It is easy to implement a never-ending infinite loop. Infinite loops might occur if the termination criterion is never met, like

```
while True:
    print("Oh no!")
```

Consider the following example:

[63]:
```
counter = 1

while counter != 5:
    counter = counter*2
    print(counter)

    # We include this to prevent the loop
    # to run infinitely long:
    if counter == 128:
        break
```

```
2
4
8
16
32
64
128
```

Another problem which is quickly implemented but overlooked is not using an iterable for loops:

[64]:
```
colours = ['green', 'red', 'blue', 'yellow']

for count in len(colours):
    print("For {} use colour {}.".format(count+1, colours[count]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[64], line 3
      1 colours = ['green', 'red', 'blue', 'yellow']
----> 3 for count in len(colours):
      4     print("For {} use colour {}.".format(count+1, colours[count]))

TypeError: 'int' object is not iterable
```

[65]:
```
colours = ['green', 'red', 'blue', 'yellow']
```

```
for count in range(len(colours)):
    print("For {} use colour {}.".format(count+1, colours[count]))
```

```
For 1 use colour green.
For 2 use colour red.
For 3 use colour blue.
For 4 use colour yellow.
```

1.7 Exercise section

(1.) Write a loop over the elements of the list

[ ]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus',␣
  ↪'Neptune']
```

which prints out only those elements whose first letter is either M or N. You can use either `for` or `while`. However, which of the two makes more sense in this task? Put your solution in the following cell:

[ ]:
```

```

(2.) Identify / print for the following dictionary

[ ]:
```
country_codes = {'Switzerland': 41, 'France': 33, 'Italy': 39, 'UK': 44,␣
  ↪'Germany': 49}
```

[ ]:
```

```

which country has the country code 39 using a `for` loop. Put your solution in the following cell:

(3.) Write a list comprehension in which the numbers from 1 to 4 are appended to the string `'05-11-19_run-'` such that the resulting list looks like

`['05-11-19_run-1','05-11-19_run-2','05-11-19_run-3','05-11-19_run-4']`

Use the `format` method for strings if possible! Put your solution in the following cell:

[ ]:
```

```

1.8 Proposed Solutions

(1.) Write a loop over the elements of the list

[1]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus',␣
  ↪'Neptune']
```

which prints out only those elements whose first letter is either M or N. You can use either `for` or `while`. However, which of the two makes more sense in this task? Put your solution in the following cell:

[2]:
```
for p in planets:
    if p[0] == 'M' or p[0] == 'N':
        print(p)
```

```
Mercury
Mars
Neptune
```

Here, we make use that we can index strings, like this:

[3]:
```
p = planets[0]
for i in range(len(p)):
    print(i,p[i])
```

```
0 M
1 e
2 r
3 c
4 u
5 r
6 y
```

[4]:
```
while planets:
    p = planets.pop()

    if p[0] == 'M' or p[0] == 'N':
#     if p[0] in ['M','N']:
        print(p)
```

```
Neptune
Mars
Mercury
```

[6]:
```
print(planets)
```

```
[]
```

(2.) Identify / print for the following dictionary

[10]:
```
country_codes = {'Switzerland': 41, 'France': 33, 'Italy': 39, 'UK': 44,␣
 ↪'Germany': 49}
```

which country has the country code 39 using a `for` loop. Put your solution in the following cell:

[8]:
```
for key,value in country_codes.items():
    if value == 39:
        print(key)
```

```
Italy
```

(3.) Write a list comprehension in which the numbers from 1 to 4 are appended to the string `'05-11-19_run-'` such that the resulting list looks like

```
['05-11-19_run-1','05-11-19_run-2','05-11-19_run-3','05-11-19_run-4']
```

Use the `format` method for strings if possible! Put your solution in the following cell:

[20]:
```python
[ '05-11-19_run-{}'.format(num) for num in range(1,5) ]
```

[20]:
```
['05-11-19_run-1', '05-11-19_run-2', '05-11-19_run-3', '05-11-19_run-4']
```

5-Plotting

October 4, 2023

1 5. Plotting

In the fifth section we have a look at visualisation and plotting in Python. In this section wie foucs on **Matplotlib** which is the standard library for plotting in Python. A popular alternative is **seaborn** which can conveniently be combined with the Pandas library and therefore will be considered in a following section.

Here, we will cover some of the basics for plotting. If you want to learn more on the different plot types like contour-plots or 3D plots, have a look at these example plots. For more in-depth tutorials check the Matplotlib tutorial overview.

In this part we learn

- how to import libraries and
- about different types of plots.

Keywords: `import`, `matplotlib.pyplot`, `plt.plot`, `plt.savefig`, `plt.scatter`, `plt.bar`, `plt.barh`, `plt.hist`, `plt.hist2d`, `plt.imshow`, `plt.subplots`, `plot_surface`

1.1 Import a library

In order to import a library, sometimes also referred to as module, the library needs to be installed and imported with the `import` keyword.

```
import matplotlib
```

Often, libraries comprise a great variety of sub-modules which provide a particular functionality. For plotting, the module would be `pyplot`.

```
import matplotlib.pyplot as plt
```

It is common to use an abbreviation for libraries. Instead of always writing the full library name whenever some functionality of the library is used, we can use `plt` if the library was imported with the keyword `as` + abbreviation.

Another way to import only as specific sub-module uses the `from` keyword.

```
from matplotlib import table
```

For example, the `table` module provides additional functionality to add a table to a plot.

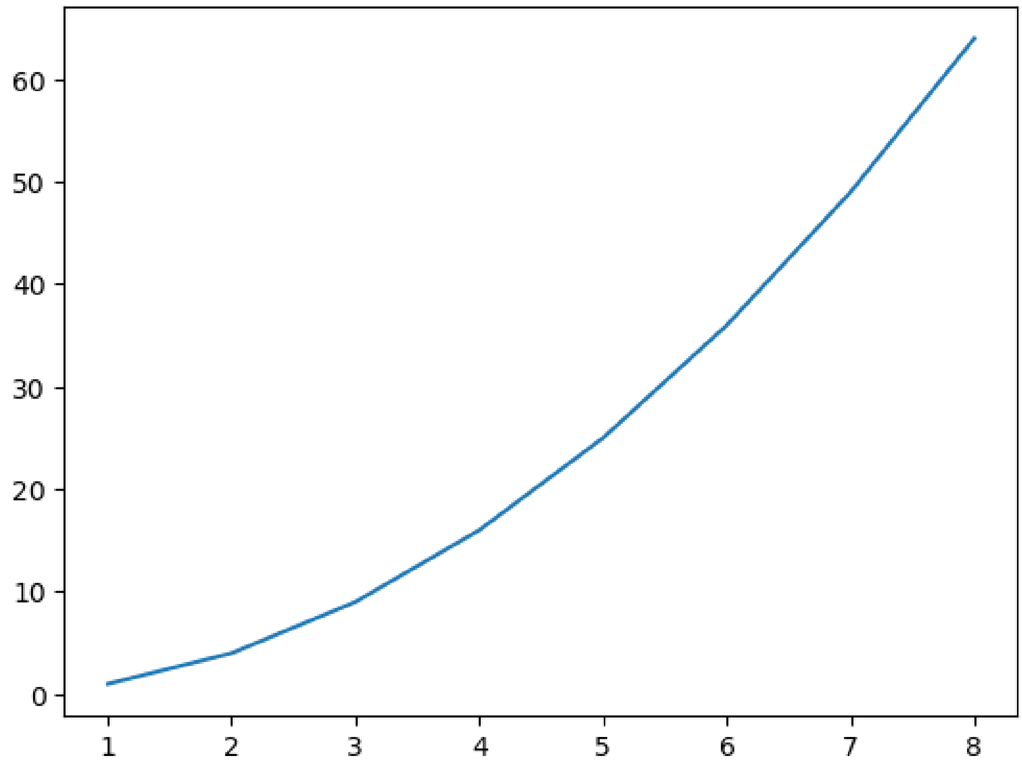
1.2 Matplotlib

Let us import the pyplot module and do some first plots.

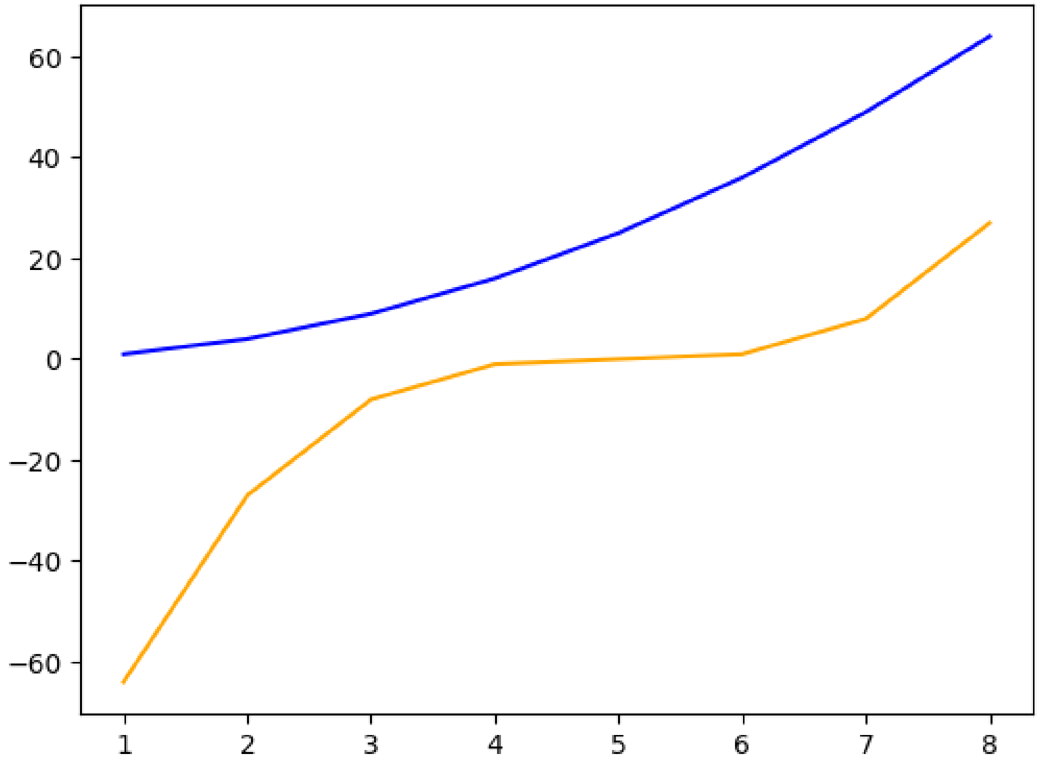
[1]:
```
import matplotlib.pyplot as plt
```

[2]:
```
x = [1,2,3,4,5,6,7,8]
y = [1,4,9,16,25,36,49,64]
```

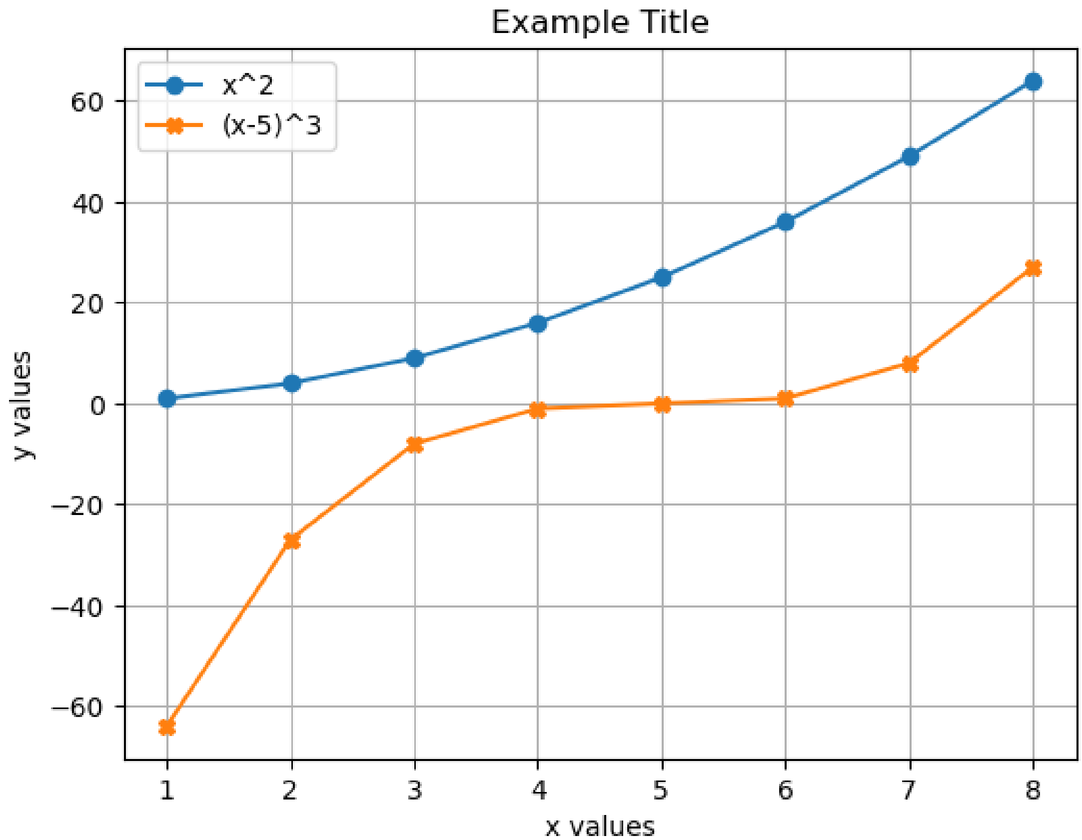
[5]:
```
plt.plot(x,y)
plt.show()
```



[6]:
```
z = [-64,-27,-8,-1,0,1,8,27]

plt.plot(x,y, color='blue')
plt.plot(x,z, color='orange')

plt.show()
```

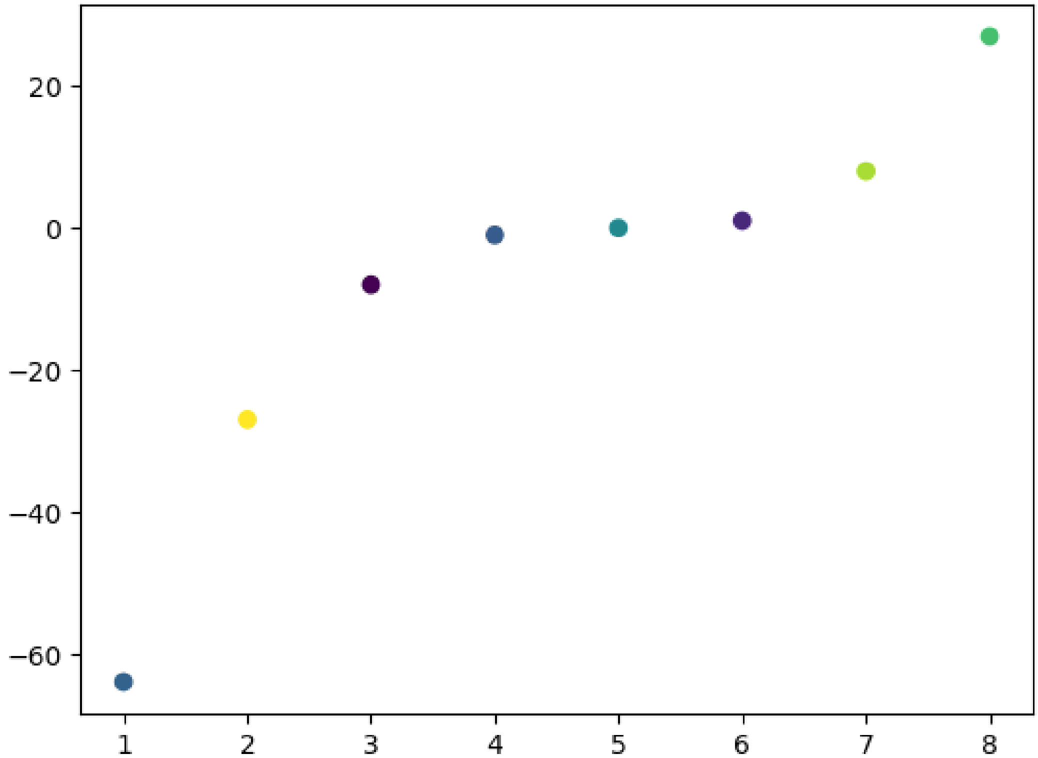


[11]:
```
plt.plot(x, y, label = 'x^2', marker = 'o')
plt.plot(x, z, label = '(x-5)^3', marker = 'X')

plt.xlabel('x values')
plt.ylabel('y values')

plt.title('Example Title')

plt.legend()

plt.grid()

plt.show()
```



[12]:
```
random_colours = [0.404, 0.994, 0.138, 0.383, 0.537, 0.238, 0.883, 0.742]

plt.scatter(x,z, c=random_colours)

plt.savefig("data/saved_plot.png", dpi = 150)
print("Plot saved to file!")
```

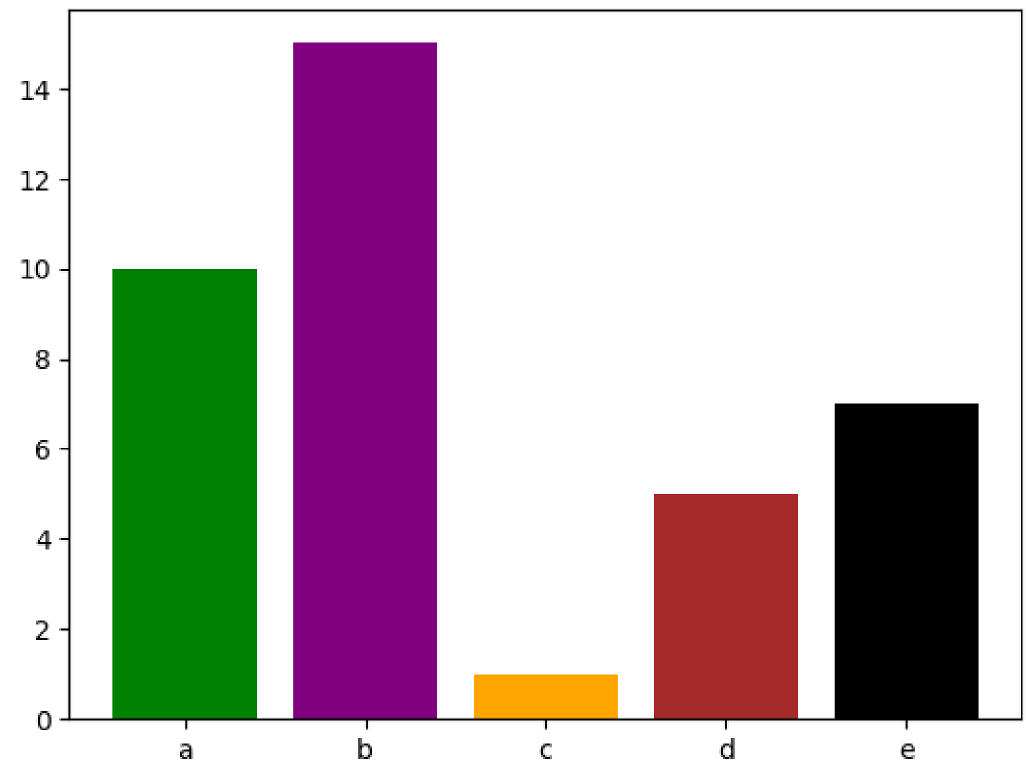
```
Plot saved to file!
```



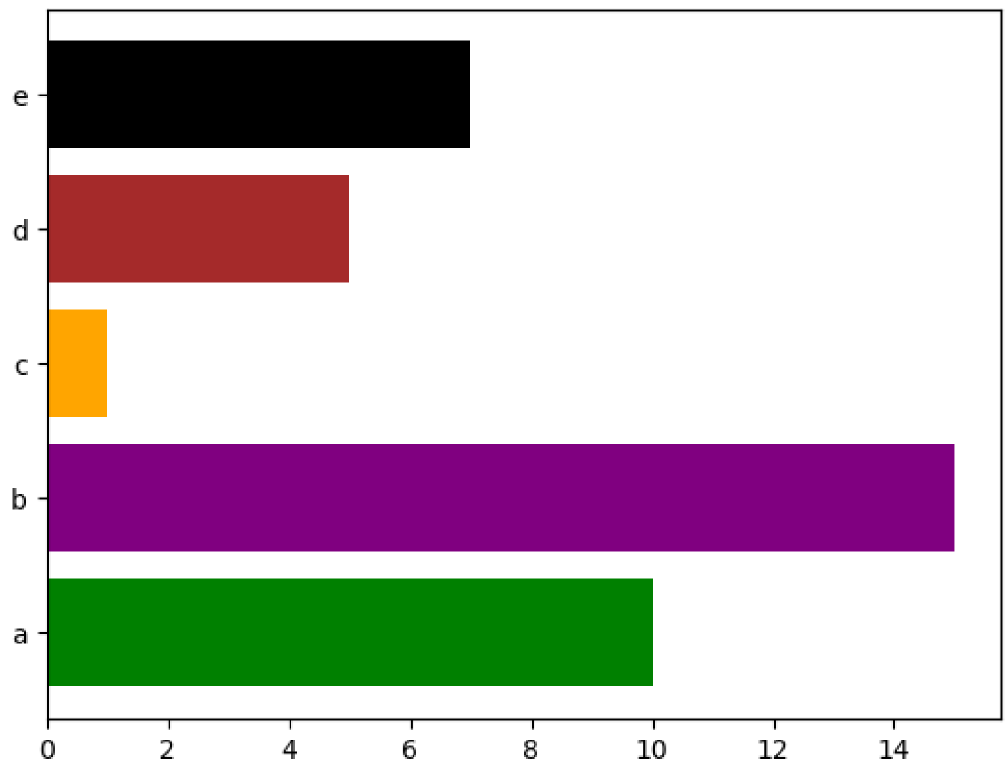
Note that there are different ways of encoding colours. There are built-in colour names like `'red'`, you can use HTML hex string `'#FFFFFF'` or values between 0 and 1 and some other ways.

Saving a plot can be done with `plt.savefig`.

[13]:
```
categories = ['a', 'b', 'c', 'd', 'e']
count = [10, 15, 1, 5, 7]

our_colours = ['green','purple','orange','brown','black']

plt.bar(categories, count, color=our_colours)
plt.show()
```



[14]:
```
plt.barh(categories, count, color=our_colours)
plt.show()
```



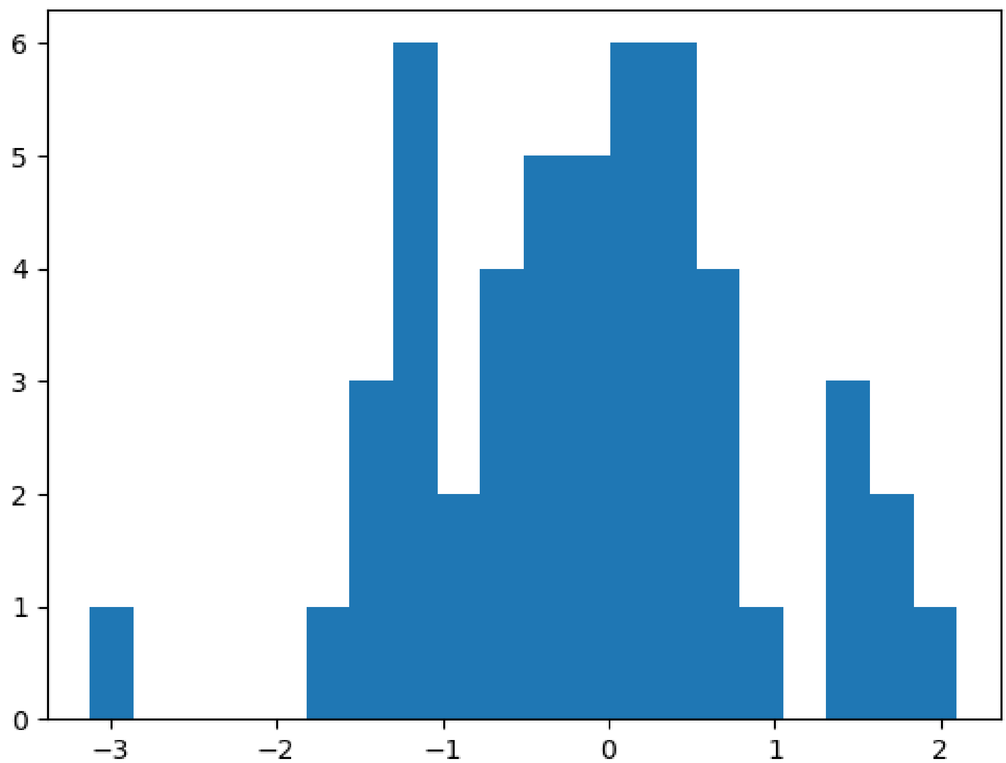
In the following we have two lists with 50 random numbers drawn from a standard normal distribution $\mathcal{N}(\mu = 0, \sigma = 1)$.

[15]:
```
random_normal_1 = [0.6612, -0.5364, -0.4819,  0.3127, -3.1218, -1.3585, -0.1426,
                   -0.217 , -0.0329,  1.8321,  0.3524,  0.2366,  1.4785,  1.
 ↪5543,
                   -1.1106,  0.8238,  0.6653,  0.7148, -1.2308,  2.0954, -0.
 ↪6032,
                   1.6136, -0.254 , -1.5159, -0.1172,  0.4954, -0.7026,  0.3951,
                   -1.1233, -1.7156, -0.2711,  0.0862, -1.0905,  0.1125, -0.
 ↪9536,
                   0.0502, -0.4832,  1.3339,  0.4802, -0.7497, -1.3208,  0.3322,
                   -0.2666,  0.6362, -1.1183,  0.2645,  0.0931, -1.1962, -0.
 ↪8989,
                   -0.0454]

random_normal_2 = [-1.1317, -0.6985, -1.2833, -1.0344,  0.8587, -1.1286,  0.
 ↪4251,
                   1.0161, -3.1386, -0.492 ,  1.6102,  0.474 , -2.3115, -0.4532,
                   -0.8317, -0.2631, -0.2534,  0.1046, -0.3384, -1.148 , -0.
 ↪5277,
```

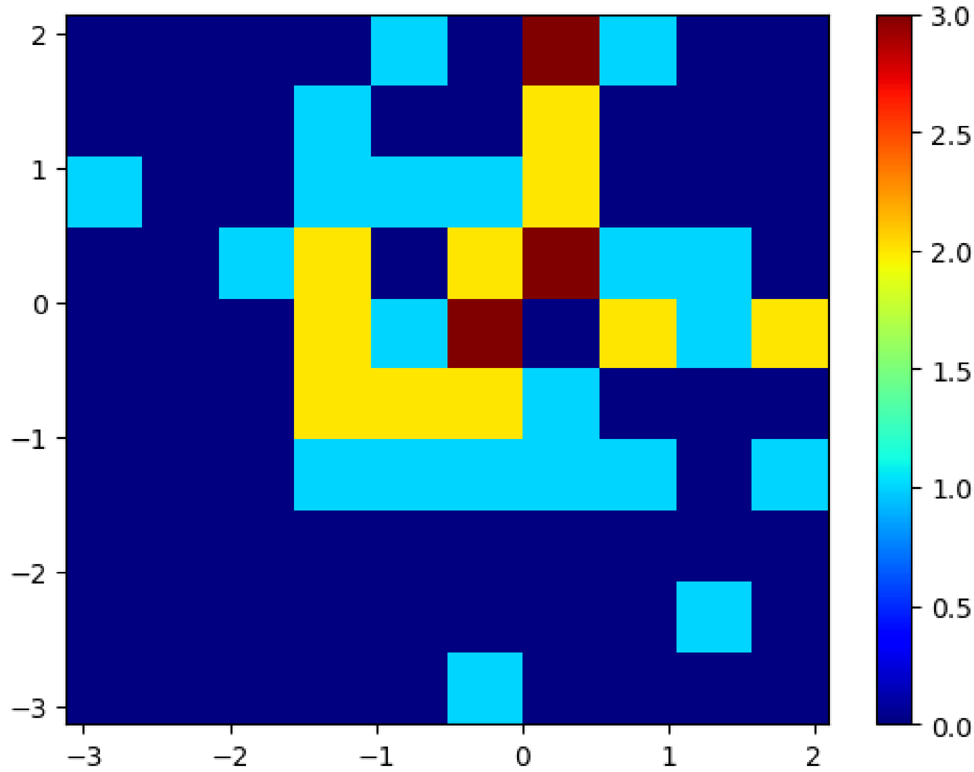
```
                   -0.1208,  0.1743, -0.6389, -0.0851,  1.6269,  0.7295,  0.
↪3874,
                   0.1361,  0.1586, -0.0874, -0.7312,  1.2467,  0.7123, -1.1726,
                   0.439 , -0.5257,  0.4521,  0.8294,  2.1029,  0.511 ,  2.141 ,
                   -0.6132,  2.0561, -0.0245,  1.6829,  1.1089,  0.9553, -0.
↪4413,
                   -0.4629]
```

We want to display the distribution of the random samples with a **histogram** and aggregate the data in 20 bins.

[18]:
```
plt.hist(random_normal_1, bins=20)
plt.show()
```

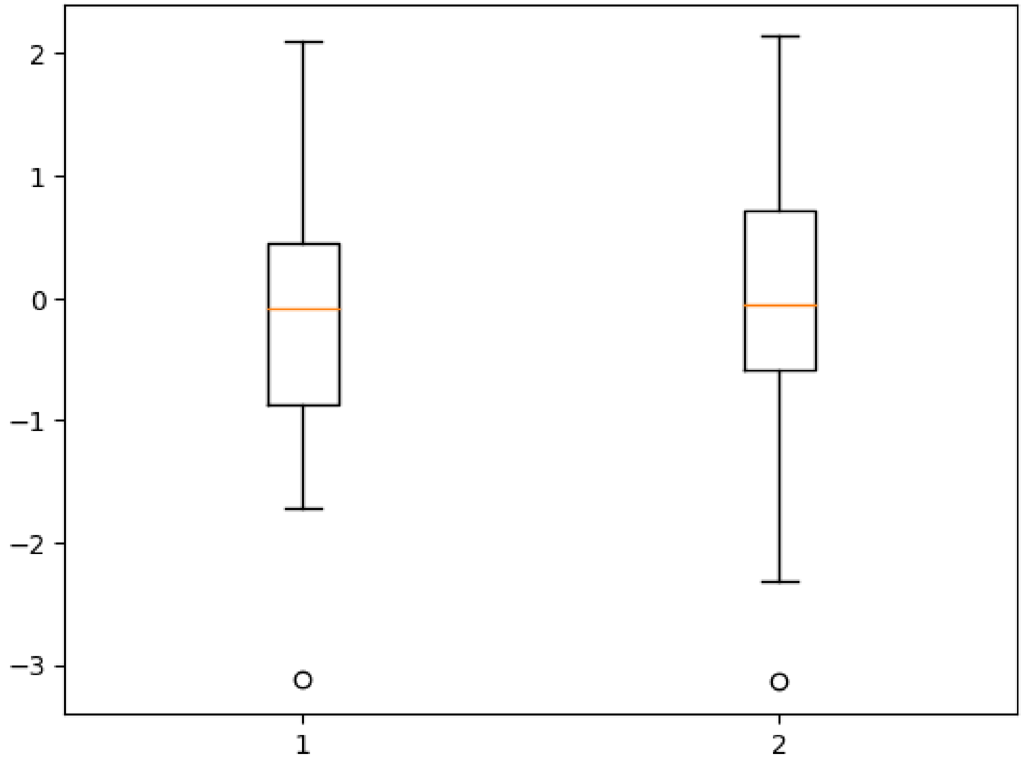


[19]:
```
plt.hist2d(random_normal_1, random_normal_2, cmap='jet', bins=10)
plt.colorbar()
plt.show()
```



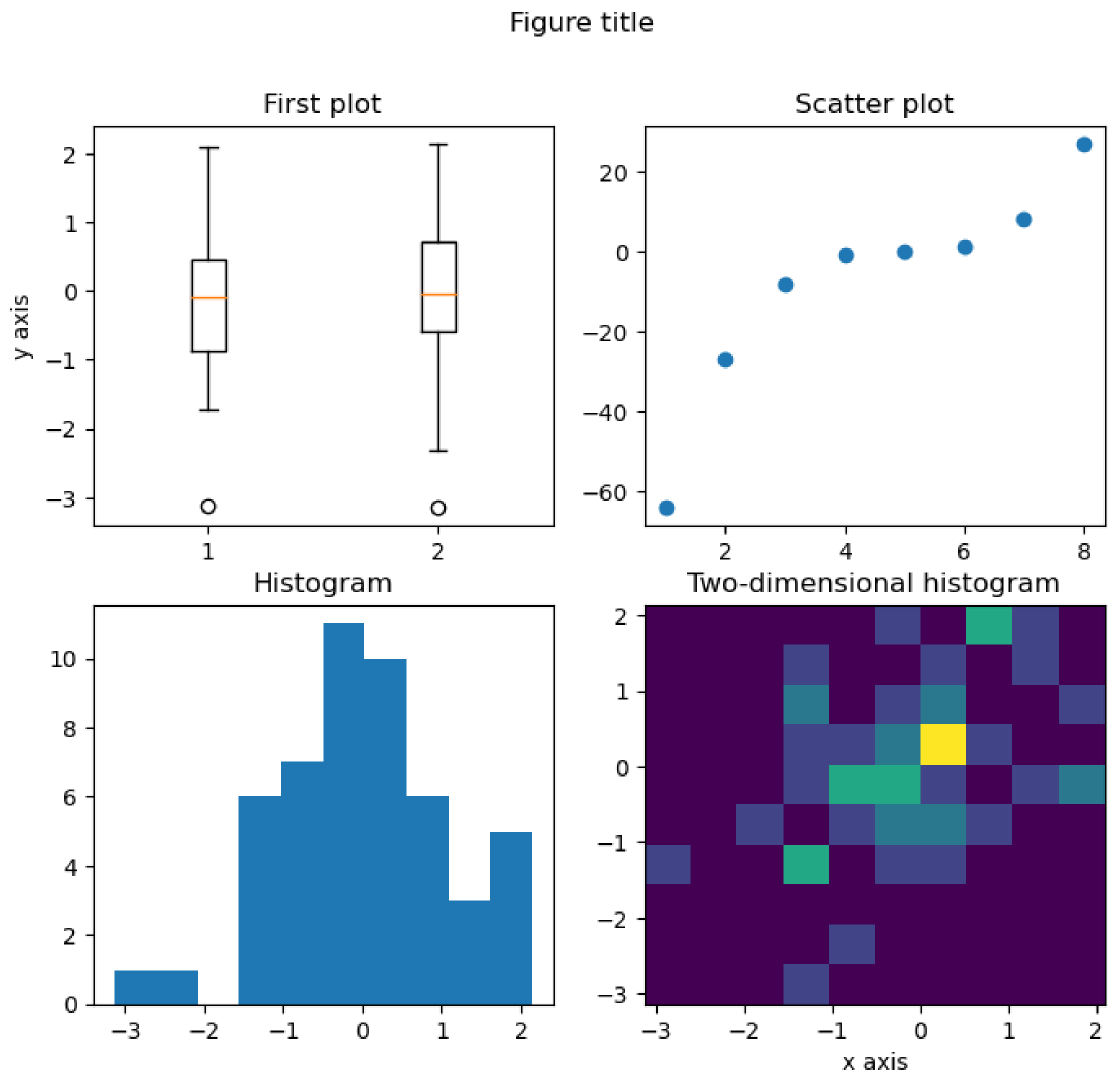
Another way to study the distribution of our random samples are **boxplots**.

[20]:
```
plt.boxplot( [random_normal_1, random_normal_2] )
plt.show()
```



Create a figure containing multiple plots with `subplots`.

[52]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 7) )

axs[0, 0].boxplot([random_normal_1, random_normal_2])
axs[0, 1].scatter(x, z)
axs[1, 0].hist(random_normal_2)
axs[1, 1].hist2d(random_normal_1, random_normal_2)

axs[0, 0].set_title("First plot")
axs[0, 1].set_title("Scatter plot")
axs[1, 0].set_title("Histogram")
axs[1, 1].set_title("Two-dimensional histogram")

axs[0, 0].set_ylabel("y axis")
axs[1, 1].set_xlabel("x axis")

fig.suptitle("Figure title")

plt.show()
```

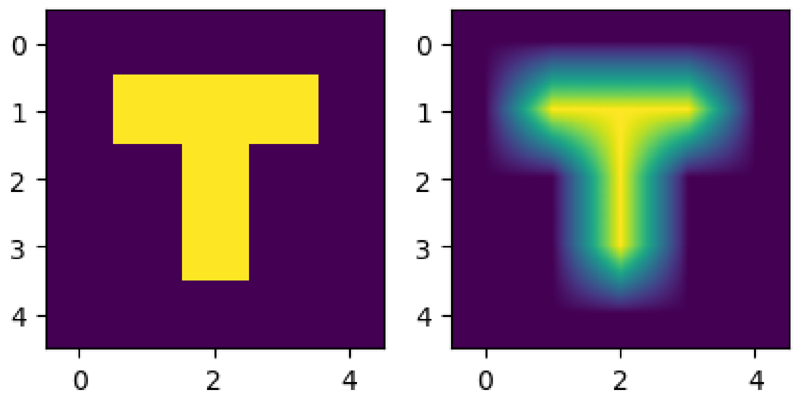


Note that setting labels for subplots with is slightly different from setting labels for individual figures. In the former case, we do `ax.set_xlabel(...)` while in the latter it was `plt.xlabel(...)`. There more such differences for subplots.

Visualise a matrix with `imshow`.

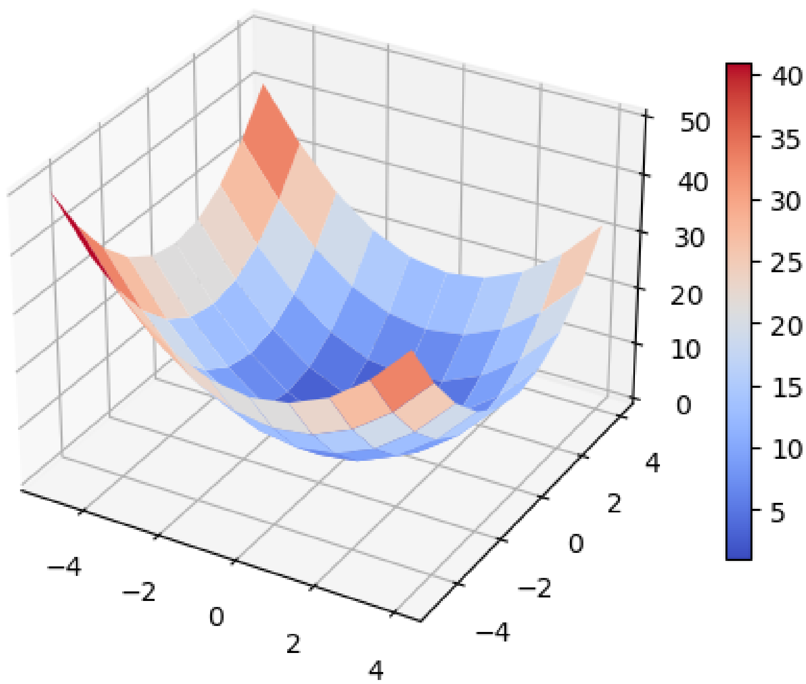
[22]:
```
matrix = [[0,0,0,0,0],
          [0,1,1,1,0],
          [0,0,1,0,0],
          [0,0,1,0,0],
          [0,0,0,0,0]]


fig, axs = plt.subplots(1,2, figsize=(5,3))
```

```
axs[0].imshow(matrix)
axs[1].imshow(matrix, interpolation='bilinear')

plt.show()
```



The following example of a **surface plot** requires some additional libraries.

[23]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

X = [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4]
Y = [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4]

X, Y = np.meshgrid(X, Y)

Z = X**2 + Y**2

surface = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)

fig.colorbar(surface, shrink=0.7)

plt.show()
```



1.3 Exercise section

(1.) Recreate the plot below by 1. sorting `random_normal_1` in ascending order and then 2. plotting `random_normal_2` (on the y-axis) against the sorted `random_normal_1` (on the x-axis).

Note that you need to set the colour, label and markers as well as the axes labels and the title. Furthermore, remember that we discussed previously how to sort list entries.

You can reassign the random variables in the next cell:

[24]:
```
import matplotlib.pyplot as plt

random_normal_1 = [0.6612, -0.5364, -0.4819,  0.3127, -3.1218, -1.3585, -0.1426,
                   -0.217 , -0.0329,  1.8321,  0.3524,  0.2366,  1.4785,  1.
 ↪5543,
                   -1.1106,  0.8238,  0.6653,  0.7148, -1.2308,  2.0954, -0.
 ↪6032,
                   1.6136, -0.254 , -1.5159, -0.1172,  0.4954, -0.7026,  0.3951,
                   -1.1233, -1.7156, -0.2711,  0.0862, -1.0905,  0.1125, -0.
 ↪9536,
                   0.0502, -0.4832,  1.3339,  0.4802, -0.7497, -1.3208,  0.3322,
```

```
                    -0.2666,  0.6362, -1.1183,  0.2645,  0.0931, -1.1962, -0.
 ↪8989,
                    -0.0454]

random_normal_2 = [-1.1317, -0.6985, -1.2833, -1.0344,  0.8587, -1.1286,  0.
 ↪4251,
                    1.0161, -3.1386, -0.492 ,  1.6102,  0.474 , -2.3115, -0.4532,
                    -0.8317, -0.2631, -0.2534,  0.1046, -0.3384, -1.148 , -0.
 ↪5277,
                    -0.1208,  0.1743, -0.6389, -0.0851,  1.6269,  0.7295,  0.
 ↪3874,
                    0.1361,  0.1586, -0.0874, -0.7312,  1.2467,  0.7123, -1.1726,
                    0.439 , -0.5257,  0.4521,  0.8294,  2.1029,  0.511 ,  2.141 ,
                    -0.6132,  2.0561, -0.0245,  1.6829,  1.1089,  0.9553, -0.
 ↪4413,
                    -0.4629]
```

Put your solution here:

[ ]:

```

```

1.4 Proposed Solutions

[42]:

```
import matplotlib.pyplot as plt
```

(1.) Recreate the plot below by 1. sorting `random_normal_1` in ascending order and then 2. plotting `random_normal_2` (on the y-axis) against the sorted `random_normal_1` (on the x-axis).

Note that you need to set the colour, label and markers as well as the axes labels and the title. Furthermore, remember that we discussed previously how to sort list entries.

You can reassign the random variables in the next cell:

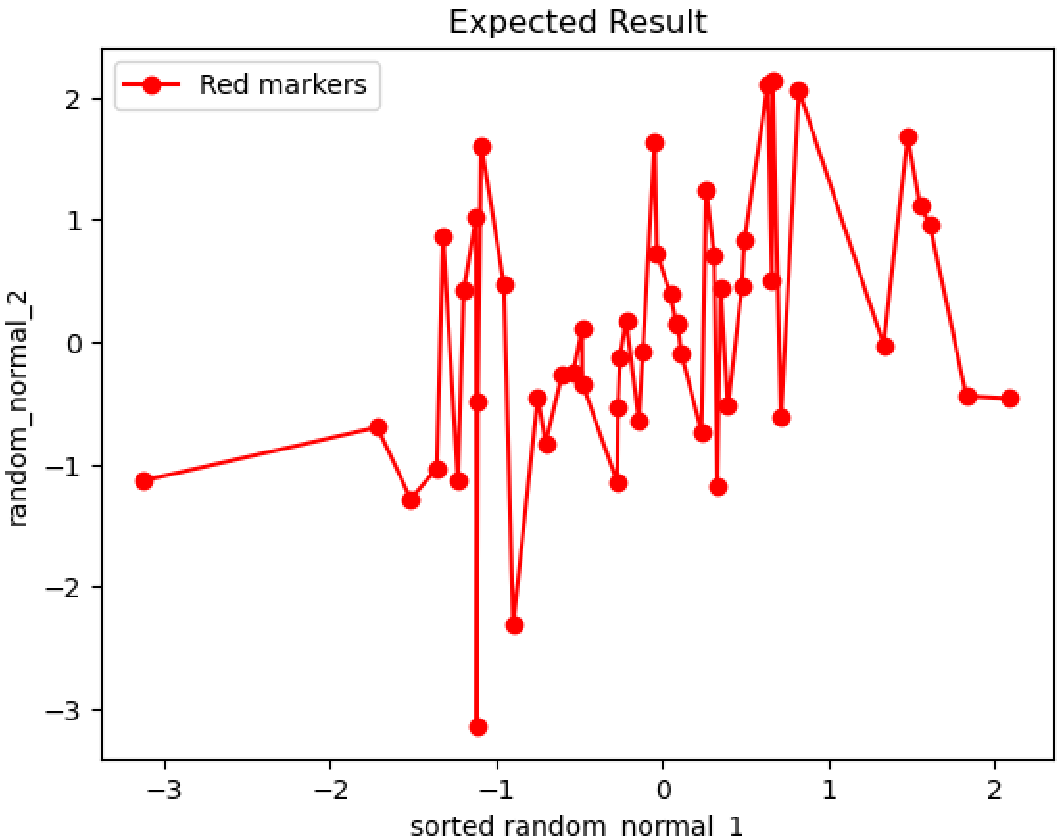
[43]:

```
random_normal_1 = [0.6612, -0.5364, -0.4819,  0.3127, -3.1218, -1.3585, -0.1426,
                    -0.217 , -0.0329,  1.8321,  0.3524,  0.2366,  1.4785,  1.
 ↪5543,
                    -1.1106,  0.8238,  0.6653,  0.7148, -1.2308,  2.0954, -0.
 ↪6032,
                    1.6136, -0.254 , -1.5159, -0.1172,  0.4954, -0.7026,  0.3951,
                    -1.1233, -1.7156, -0.2711,  0.0862, -1.0905,  0.1125, -0.
 ↪9536,
                    0.0502, -0.4832,  1.3339,  0.4802, -0.7497, -1.3208,  0.3322,
                    -0.2666,  0.6362, -1.1183,  0.2645,  0.0931, -1.1962, -0.
 ↪8989,
                    -0.0454]
```

```
random_normal_2 = [-1.1317, -0.6985, -1.2833, -1.0344,  0.8587, -1.1286,  0.
↪4251,
                   1.0161, -3.1386, -0.492 ,  1.6102,  0.474 , -2.3115, -0.4532,
                  -0.8317, -0.2631, -0.2534,  0.1046, -0.3384, -1.148 , -0.
↪5277,
                  -0.1208,  0.1743, -0.6389, -0.0851,  1.6269,  0.7295,  0.
↪3874,
                   0.1361,  0.1586, -0.0874, -0.7312,  1.2467,  0.7123, -1.1726,
                   0.439 , -0.5257,  0.4521,  0.8294,  2.1029,  0.511 ,  2.141 ,
                  -0.6132,  2.0561, -0.0245,  1.6829,  1.1089,  0.9553, -0.
↪4413,
                  -0.4629]
```

Put your solution here:

[48]:
```
random_normal_1.sort()

plt.plot(random_normal_1,random_normal_2, color='red',
         label='Red markers', marker='o')

plt.xlabel('sorted random_normal_1')
plt.ylabel('random_normal_2')

plt.title('Expected Result')

plt.legend()

#plt.show()
plt.savefig("images/plot_exercise_result.png", bbox_inches='tight')
```


Expected Result
Red markers
random_normal_2
sorted random_normal_1
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2

6-Numpy_Pandas

October 4, 2023

1 6. NumPy & Pandas

In the sixth section we present two powerful libraries for efficient scientific computing: **NumPy** and **Pandas**. Note that we won't be able to illustrated the full potential of each library but rather present a selection of useful tools. For more details you might want to check out the NumPy User Guide or the Pandas Tutorial Guide.

In addition, we have a look on how one can use **seaborn** to create plots when working with Pandas. For more details on seaborn you might want to visit the seaborn Tutorial Overview.

In this part we revisit some of the previous concepts and learn

- about array manipulation with NumPy,
- how to work with Pandas DataFrames and
- how to plot in these frameworks.

Keywords: `np.array`, `shape`, `astype`, `np.matmul`, `np.multiply`, `np.mean`, `np.arange`, `np.reshape`, `np.append`, `np.random`, `np.newaxis`, `np.savetxt`, `np.loadtxt`, `pd.DataFrame`, `value_counts`, `head`,
`pd.groupby`, `pd.describe`, `pd.read_csv`, `pd.to_csv`, `seaborn`, `sns.countplot`, `sns.boxplot`, `sns.violinplot`, `sns.jointplot`

1.1 NumPy

NumPy adds a lot of efficient ways to work with large list and matrices, which are also referred to as **arrays**, and a large number of high-level mathematical functions. In most cases, it is much **more efficient** to work with NumPy objects instead of the built-in objects we encountered so far. So if you have large arrays you are working with, performing calculations with NumPy is usally a good choice to do fast computation.

As before, we need to import the NumPy library first. The common abbrevation is `np`.

[1]:
```
import numpy as np
```

1.1.1 Initialising NumPy arrays

Let us compare the "old" list type with NumPy arrays!

[2]:
```python
old_list = [1,2,3,4]
np_list = np.array( old_list )

print(old_list, "vs", np_list)
```

```
[1, 2, 3, 4] vs [1 2 3 4]
```

[3]:
```python
old_matrix = [[1,2,3],[4,5,6]]
np_matrix = np.array(old_matrix)

print(old_matrix, "\nvs\n", np_matrix)
```

```
[[1, 2, 3], [4, 5, 6]]
vs
 [[1 2 3]
 [4 5 6]]
```

[4]:
```python
print(type(old_matrix))
print(type(np_matrix))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

[5]:
```python
len(np_matrix)
```

[5]: 2

[6]:
```python
np_matrix.shape
```

[6]: (2, 3)

[8]:
```python
np_matrix.ndim
```

[8]: 2

Note that the output of `shape` can be interpeted as the **number of rows and columns** in the matrix, while `dim` specifies the number of **dimensions**, i.e. *there is one row and one column dimension.*

[9]:
```python
test_array = np.arange(2,10)
print("array:", test_array)
print("shape:", test_array.shape)
print("number dimensions:", test_array.ndim)
```

```
array: [2 3 4 5 6 7 8 9]
shape: (8,)
number dimensions: 1
```

[10]:
```
matrix = np.array( [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ] )
print("array:\n", matrix)
print("shape:", matrix.shape)
print("number of dimensions:", matrix.ndim)
```

```
array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
shape: (3, 3)
number of dimensions: 2
```

[11]:
```
matrix[0]
```

[11]:
```
array([1, 2, 3])
```

[12]:
```
matrix[1,1]
```

[12]:
```
5
```

[13]:
```
matrix[1][1]
```

[13]:
```
5
```

[14]:
```
tensor = np.array([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]],
    [[13, 14, 15], [16, 17, 18]]
])

print("array:\n", tensor)
print("shape:", tensor.shape)
print("number of dimensions:", tensor.ndim)
```

```
array:
 [[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]

 [[13 14 15]
  [16 17 18]]]
shape: (3, 2, 3)
number of dimensions: 3
```

Note that you can think of a tensor as a 3-dimension matrix or you can view it as a "cube of values".

Some special types of arrays:

[15]:
```
np.zeros( (3,4) )
```

[15]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

[20]:
```
np.ones( (3,4) )
```

[20]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

[21]:
```
eye = np.eye( 3 )
print(eye)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

1.1.2 Type converison

As before, changing types is not a big deal in Python. However, note that we now use new functions for this.

[32]:
```
matrix
```

[32]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[35]:
```
matrix = matrix.astype(float)
```

[36]:
```
matrix
```

[36]:
```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

Note that we used `float(...)` to convert a variable to a float variable, before.

[25]:
```
type(matrix.astype(float))
```

[25]:
```
numpy.ndarray
```

[26]:
```
matrix.dtype
```

[26]: dtype('float64')

Note that similar to `.astype(...)`, we require `.dtype` with NumPy arrays.

[37]:
```
matrix.astype(str)
```

[37]:
```
array([['1.0', '2.0', '3.0'],
       ['4.0', '5.0', '6.0'],
       ['7.0', '8.0', '9.0']], dtype='<U32')
```

1.1.3 Arithmetic

Arithmetic operations work, again, very intuitively. But note that some operations with NumPy arrays are much closer to the mathematical intuition than with lists.

[38]:
```
old_matrix
```

[38]: [[1, 2, 3], [4, 5, 6]]

[39]:
```
old_matrix*2
```

[39]: [[1, 2, 3], [4, 5, 6], [1, 2, 3], [4, 5, 6]]

Note that the "rows" of `old_matrix` just got append when using `*2`. With NumPy arrays, we can indeed multiply the matrix with the scalar 2.

[40]:
```
matrix
```

[40]:
```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

[41]:
```
matrix*2
```

[41]:
```
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.],
       [14., 16., 18.]])
```

[42]:
```
matrix - matrix*2
```

[42]:
```
array([[-1., -2., -3.],
       [-4., -5., -6.],
       [-7., -8., -9.]])
```

[43]:
```
print(matrix)

matrix_mulitplication = np.matmul(matrix,matrix)
```

```
print(matrix_mulitplication)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
[[ 30.  36.  42.]
 [ 66.  81.  96.]
 [102. 126. 150.]]
```

Note that `np.matmul` provides the matrix multiplication which we would expect, where entry

[44]:
```
matrix_mulitplication[0][0]
```

[44]:
```
30.0
```

is calculated with

$$1 \cdot 1 + 2 \cdot 4 + 3 \cdot 7 = 1 + 8 + 21 = 30.$$

However,

[45]:
```
elementwise_product = matrix * matrix
print(elementwise_product)
```

```
[[ 1.  4.  9.]
 [16. 25. 36.]
 [49. 64. 81.]]
```

is obtained by multiplying each entry of the first matrix with the respective entry of the second matrix element-wise. I.e.

[46]:
```
elementwise_product[2][2]
```

[46]:
```
81.0
```

is calculated with `matrix[2,2]` $= 9$ times `matrix[2,2]` $= 9$, so

$$9 \cdot 9 = 81.$$

In NumPy this is also implemented as

[47]:
```
np.multiply(matrix, matrix)
```

[47]:
```
array([[ 1.,  4.,  9.],
       [16., 25., 36.],
       [49., 64., 81.]])
```

[48]:
```
matrix_mulitplication = matrix @ matrix
print(matrix_mulitplication)
```

```
[[ 30.  36.  42.]
 [ 66.  81.  96.]
 [102. 126. 150.]]
```

Note that operator `@` can be used for the matrix multiplication of numpy arrays (insted of np.matmul).

[49]:
```
vec1 = np.array([2,2,2])
vec2 = np.array([5,10,20])

np.dot(vec1, vec2)
```

[49]:
```
70
```

Note that `np.dot` is the scalar or dot product.

[50]:
```
np_exp = np.exp(2)
print(np_exp)
```

```
7.38905609893065
```

[51]:
```
2.71828**2
```

[51]:
```
7.3890461584
```

[52]:
```
np.power(2,4)
```

[52]:
```
16
```

[53]:
```
np.sqrt(16)
```

[53]:
```
4.0
```

[54]:
```
np.log(np_exp)
```

[54]:
```
2.0
```

1.1.4 Common array operations

[55]:
```
matrix
```

[55]:
```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

[56]:
```
matrix.transpose()
```

```
[56]: array([[1., 4., 7.],
             [2., 5., 8.],
             [3., 6., 9.]])
```

```
[57]: matrix.flatten()
```

```
[57]: array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
[58]: matrix.min()
```

```
[58]: 1.0
```

```
[59]: matrix.max()
```

```
[59]: 9.0
```

```
[60]: matrix.mean()
```

```
[60]: 5.0
```

```
[61]: matrix
```

```
[61]: array([[1., 2., 3.],
             [4., 5., 6.],
             [7., 8., 9.]])
```

```
[62]: matrix.sum()
```

```
[62]: 45.0
```

Note that you can also perform a column-wise sum

```
[63]: matrix
```

```
[63]: array([[1., 2., 3.],
             [4., 5., 6.],
             [7., 8., 9.]])
```

```
[64]: matrix.sum(axis=0)
```

```
[64]: array([12., 15., 18.])
```

or row-wise sum

```
[65]: matrix.sum(axis=1)
```

```
[65]: array([ 6., 15., 24.])
```

```
[67]: matrix.sum(axis=2)
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
Cell In[67], line 1
----> 1 matrix.sum(axis=2)

File ~/anaconda3/envs/pythonCC/lib/python3.11/site-packages/numpy/core/_methods
 ↪py:49, in _sum(a, axis, dtype, out, keepdims, initial, where)
     47 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     48          initial=_NoValue, where=True):
---> 49     return umr_sum(a, axis, dtype, out, keepdims, initial, where)

AxisError: axis 2 is out of bounds for array of dimension 2
```

1.1.5 Reshaping, slicing and appending arrays

[68]:
```
new_np_matrix = np.arange(24)
new_np_matrix
```

[68]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23])
```

[71]:
```
new_np_matrix = new_np_matrix.reshape( 4,6 )
new_np_matrix
```

[71]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

[72]:
```
new_np_matrix.shape
```

[72]:
```
(4, 6)
```

[74]:
```
new_np_matrix = new_np_matrix.reshape(2,3,4)
new_np_matrix
```

[74]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

[75]:
```
new_np_matrix[1]
```

[75]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

[77]:
```
new_np_matrix[0,0,1]
```

[77]:
```
1
```

[82]:
```
new_np_matrix[0,0,:]
```

[82]:
```
array([0, 1, 2, 3])
```

Note that this short for

`new_np_matrix[0,0,start:end]`

and is the same as

`new_np_matrix[0,0]`

[83]:
```
new_np_matrix[0,:,0]
```

[83]:
```
array([0, 4, 8])
```

Note that : indicates that all elements in this dimension shall be selected.

[84]:
```
np_vector = np.array([10,20,30,40])

np_append1 = np.append( np_vector, new_np_matrix[0,0,:] )

print(np_append1)
print(np_vector)
```

```
[10 20 30 40  0  1  2  3]
[10 20 30 40]
```

Note that after appending to matrices the result needs to be assigned to a variable. For lists this was not necessary because the result of

`list_vector.extend(list_matrix[0,0,:])`

would be directly (*in-place*) appended to `list_vector`.

With NumPy arrays, the **dimensions of the arrays play an important role** for appending. Let's have a look at the shape of `np_vector` and `new_np_matrix[0,0,:]`:

[85]:
```
list_vector = [10,20,30,40]
list_matrix = [0,1,2,3]
print(list_vector)
```

```
[10, 20, 30, 40]
```

[86]:
```
list_vector.extend(list_matrix)
print(list_vector)
```

```
[10, 20, 30, 40, 0, 1, 2, 3]
```

[87]:
```
print("Dimension np_vector:", np_vector.shape)
print("Dimension new_np_matrix[0,0,:]:", new_np_matrix[0,0,:].shape)
```

```
Dimension np_vector: (4,)
Dimension new_np_matrix[0,0,:]: (4,)
```

[88]:
```
np_a = np.array([ [1,2,3,4] ])
np_b = np.array([ [10,12,13,14] ])
print(np_a)
```

```
[[1 2 3 4]]
```

[89]:
```
print("Dimension np_a:", np_a.shape)
print("Dimension np_b:", np_b.shape)
```

```
Dimension np_a: (1, 4)
Dimension np_b: (1, 4)
```

Note that for `np_vector` and `new_np_matrix[0,0,:]` we had vectors of length 4 and for `np_a` and `np_b` we have "matrices" with one row and 4 columns.

[90]:
```
np.append(np_a, np_b)
```

[90]:
```
array([ 1,  2,  3,  4, 10, 12, 13, 14])
```

[91]:
```
np.append(np_a, np_b).shape
```

[91]:
```
(8,)
```

[92]:
```
np.append(np_a, np_b, axis = 0)
```

[92]:
```
array([[ 1,  2,  3,  4],
       [10, 12, 13, 14]])
```

[93]:
```
np.append(np_a, np_b, axis = 0).shape
```

[93]:
```
(2, 4)
```

[94]:
```
np.append(np_a, np_b, axis = 1)
```

[94]:
```
array([[ 1,  2,  3,  4, 10, 12, 13, 14]])
```

[95]:
```
np.append(np_a, np_b, axis = 1).shape
```

[95]:
```
(1, 8)
```

[96]:
```
print(np_a)
print(np_a.shape)
```

```
[[1 2 3 4]]
(1, 4)
```

[97]:
```
print(new_np_matrix[0,0,:])
print(new_np_matrix[0,0,:].shape)
```

```
[0 1 2 3]
(4,)
```

[98]:
```
np.append(np_a, new_np_matrix[0,0,:], axis=0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[98], line 1
----> 1 np.append(np_a, new_np_matrix[0,0,:], axis=0)

File ~/anaconda3/envs/pythonCC/lib/python3.11/site-packages/numpy/lib/
↪function_base.py:5617, in append(arr, values, axis)
   5615     values = ravel(values)
   5616     axis = arr.ndim-1
-> 5617 return concatenate((arr, values), axis=axis)

ValueError: all the input arrays must have same number of dimensions, but the␣
↪array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```

[99]:
```
np.append(np_a, new_np_matrix[0,0,:].reshape(1,4), axis=0)
```

[99]:
```
array([[1, 2, 3, 4],
       [0, 1, 2, 3]])
```

[102]:
```
new_np_matrix[0,0,:].transpose()
```

[102]:
```
array([0, 1, 2, 3])
```

1.1.6 Random numbers

NumPy offers a lot of different probability distributions to sample from.

[103]:
```
np.random.random(size=4)
```

[103]:
```
array([0.97372977, 0.28046935, 0.32048414, 0.9042341 ])
```

Note that `random` provides 4 random numbers uniformly sampled between 0 and 1.

[106]:
```
np.random.randint(low=0, high=10, size=3)
```

[106]:
```
array([3, 7, 0])
```

Note that `randint` provides 3 random integers uniformly sampled between the integer specified as low (0) and high (10).

[107]:
```
np.random.randn(5)
```

[107]:
```
array([-0.9721647 ,  0.45659072, -0.37411587, -0.24889061, -0.85574767])
```

Note that `randn` denotes sampling from the standard normal distribution.

[108]:
```
np.random.normal(loc=5, scale=2.0, size=4)
```

[108]:
```
array([1.27471438, 4.16985194, 5.74368716, 6.20521112])
```

Note that `loc` corresponds to the mean μ and `scale` to the standard deviation σ of the normal / Gaussian distribution.

[110]:
```
np.random.seed(1234)
```

Note that `seed(1234)` sets the *seed* or *starting point* with index `1234` from which the (pseudo) random numbers are generated. In this way, the same sequence of (pseudo) random numbers can be retrieved. This means if we execute

[113]:
```
np.random.random(size=4)
```

[113]:
```
array([0.95813935, 0.87593263, 0.35781727, 0.50099513])
```

the first three runs will always produce

1. `array([0.19151945, 0.62210877, 0.43772774, 0.78535858])`
2. `array([0.77997581, 0.27259261, 0.27646426, 0.80187218])`
3. `array([0.95813935, 0.87593263, 0.35781727, 0.50099513])`

1.1.7 Read-in and write files

NumPy comes in really handy if we can use it for our data manipulation. Usually, this requires that we read in data from a file, first.

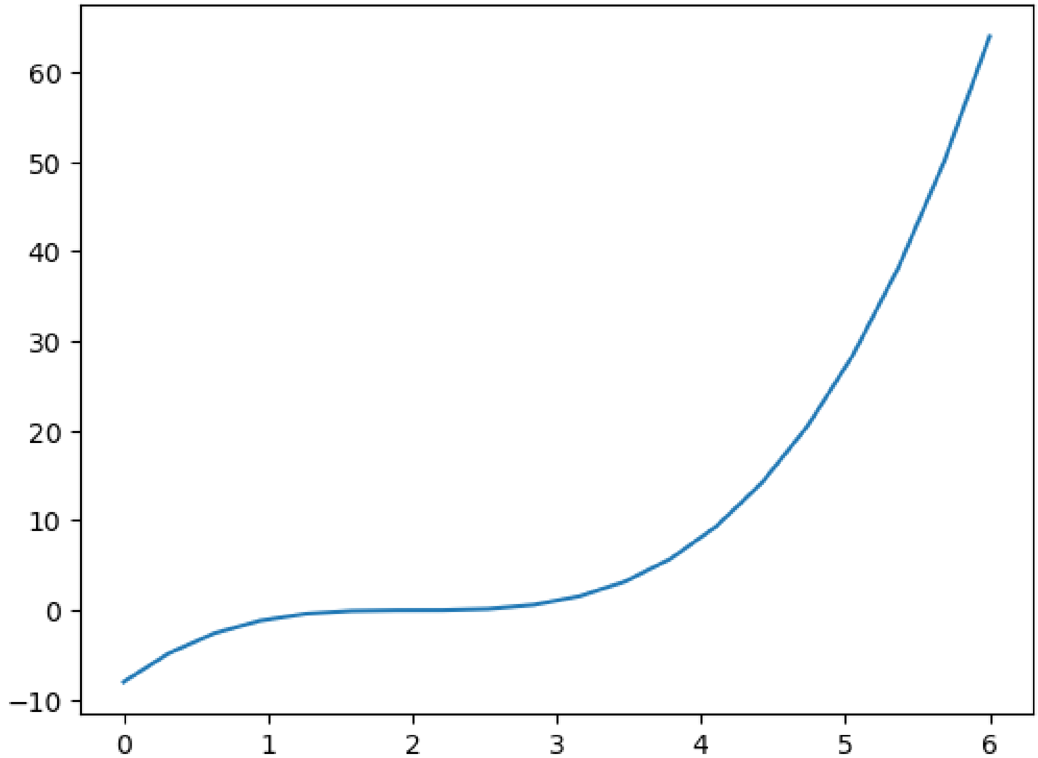
[114]:
```
csv_data = np.loadtxt('data/numpy_example.csv', delimiter = ',')

print(csv_data)
```

```
[[ 0.0000e+00 -8.0000e+00]
 [ 3.1600e-01 -4.7770e+00]
 [ 6.3200e-01 -2.5620e+00]
 [ 9.4700e-01 -1.1660e+00]
 [ 1.2630e+00 -4.0000e-01]
 [ 1.5790e+00 -7.5000e-02]
 [ 1.8950e+00 -1.0000e-03]
 [ 2.2110e+00  9.0000e-03]
 [ 2.5260e+00  1.4600e-01]
 [ 2.8420e+00  5.9700e-01]
 [ 3.1580e+00  1.5520e+00]
 [ 3.4740e+00  3.2000e+00]
 [ 3.7890e+00  5.7300e+00]
 [ 4.1050e+00  9.3310e+00]
 [ 4.4210e+00  1.4191e+01]
 [ 4.7370e+00  2.0500e+01]
 [ 5.0530e+00  2.8446e+01]
 [ 5.3680e+00  3.8219e+01]
 [ 5.6840e+00  5.0007e+01]
 [ 6.0000e+00  6.4000e+01]]
```

[115]:
```
csv_data.shape
```

[115]:
```
(20, 2)
```

[116]:
```
import matplotlib.pyplot as plt

plt.plot(csv_data[:,0], csv_data[:,1])
plt.show()
```



Let's take the cubic root of the second column with `np.cbrt` and multiply the result with `-10`.

[117]:
```
new_column = np.cbrt(csv_data[:,1]) * -10

print("new_column:\n", new_column)
print("shape:", new_column.shape)
```

```
new_column:
 [ 20.          16.84166717  13.6833691   10.52526042   7.368063
   4.21716333   1.          -2.08008382  -5.26563743  -8.42024595
 -11.57792074 -14.73612599 -17.89444393 -21.0527773  -24.21053204
 -27.36851837 -30.52626958 -33.68421571 -36.84203412 -40.        ]
shape: (20,)
```

We would like to append `new_column` to our data matrix. For this to work we need to reshape our vector of length 20 to a matrix of shape 20 x 1. We can use `np.newaxis` for this.

[118]:
```
new_column = new_column[:, np.newaxis]
# or
# new_column = new_column.reshape(20,1)

print("new_column:\n", new_column)
print("shape:", new_column.shape)
```

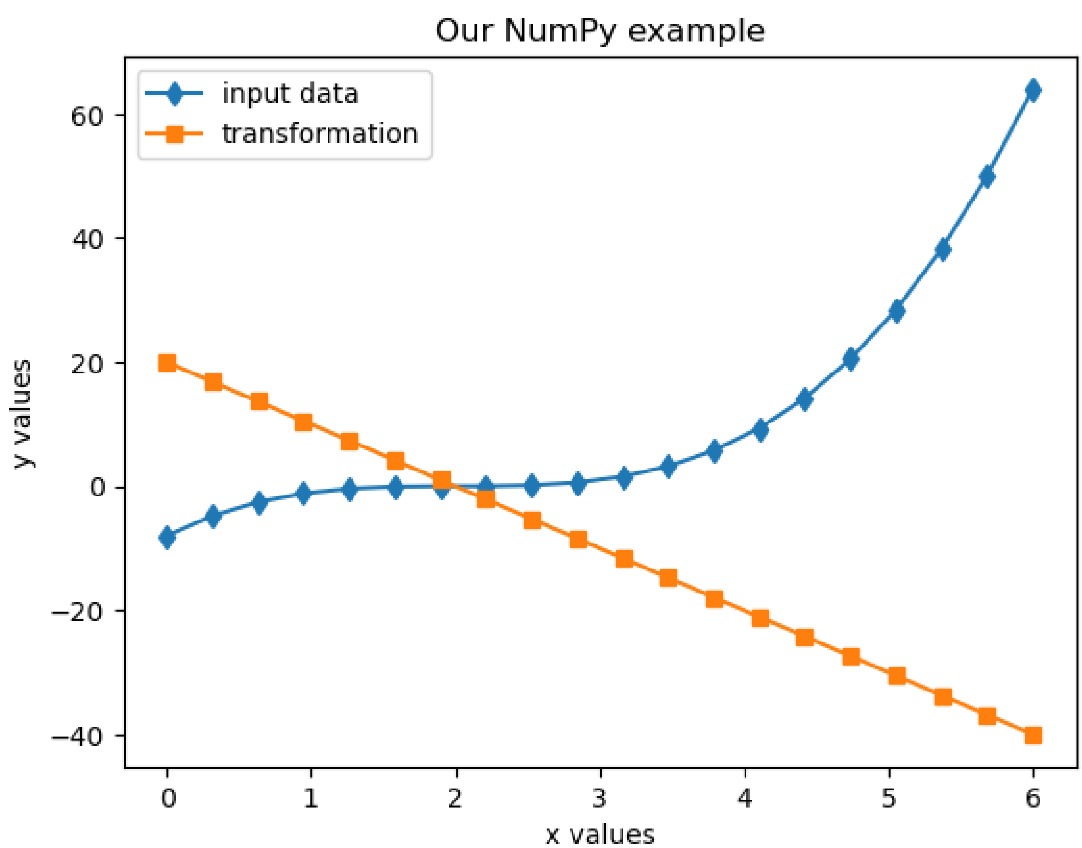
```
new_column:
 [[ 20.        ]
 [ 16.84166717]
 [ 13.6833691 ]
 [ 10.52526042]
 [  7.368063  ]
 [  4.21716333]
 [  1.        ]
 [ -2.08008382]
 [ -5.26563743]
 [ -8.42024595]
 [-11.57792074]
 [-14.73612599]
 [-17.89444393]
 [-21.0527773 ]
 [-24.21053204]
 [-27.36851837]
 [-30.52626958]
 [-33.68421571]
 [-36.84203412]
 [-40.        ]]
shape: (20, 1)
```

[119]:
```
new_csv_data = np.append(csv_data, new_column, axis = 1)
new_csv_data
```

[119]:
```
array([[ 0.00000000e+00, -8.00000000e+00,  2.00000000e+01],
       [ 3.16000000e-01, -4.77700000e+00,  1.68416672e+01],
       [ 6.32000000e-01, -2.56200000e+00,  1.36833691e+01],
       [ 9.47000000e-01, -1.16600000e+00,  1.05252604e+01],
       [ 1.26300000e+00, -4.00000000e-01,  7.36806300e+00],
       [ 1.57900000e+00, -7.50000000e-02,  4.21716333e+00],
       [ 1.89500000e+00, -1.00000000e-03,  1.00000000e+00],
       [ 2.21100000e+00,  9.00000000e-03, -2.08008382e+00],
       [ 2.52600000e+00,  1.46000000e-01, -5.26563743e+00],
       [ 2.84200000e+00,  5.97000000e-01, -8.42024595e+00],
       [ 3.15800000e+00,  1.55200000e+00, -1.15779207e+01],
       [ 3.47400000e+00,  3.20000000e+00, -1.47361260e+01],
       [ 3.78900000e+00,  5.73000000e+00, -1.78944439e+01],
       [ 4.10500000e+00,  9.33100000e+00, -2.10527773e+01],
       [ 4.42100000e+00,  1.41910000e+01, -2.42105320e+01],
       [ 4.73700000e+00,  2.05000000e+01, -2.73685184e+01],
       [ 5.05300000e+00,  2.84460000e+01, -3.05262696e+01],
       [ 5.36800000e+00,  3.82190000e+01, -3.36842157e+01],
       [ 5.68400000e+00,  5.00070000e+01, -3.68420341e+01],
       [ 6.00000000e+00,  6.40000000e+01, -4.00000000e+01]])
```

Let’s visualise the result.

[120]:
```
plt.plot(new_csv_data[:,0], new_csv_data[:,1], label = 'input data', marker =␣
 ↪'d')
plt.plot(new_csv_data[:,0], new_csv_data[:,2], label = 'transformation', marker␣
 ↪= 's')

plt.xlabel('x values')
plt.ylabel('y values')

plt.title('Our NumPy example')

plt.legend()

plt.show()
```



Finally, we store the results in a new .csv file with `np.savetxt`.

[121]:
```
np.savetxt('data/saved_numpy_example.csv', new_csv_data, delimiter=',',
           fmt='%1.3f', header='x,y,z')
```

Note that `delimiter` sets the character with which the numbers in the resulting output file shall be separated with. Further, `fmt='%1.3f'` specifies that you want your entries to be stored as floats with 3 decimals. Another example would be `fmt = '%d'` which would indicate that the entries shall be saved as integers. *** ## Pandas

Pandas is another Python library which offers a powerful way to work with more efficient data structures and allows for advanced data manipulation and analysis. If you have some experience with R, the way to work with Pandas will look very familiar to you.

The common abbrevation for Pandas is `pd`.

[122]:
```
import pandas as pd
```

[123]:
```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
matrix
```

[123]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[124]:
```
matrix_df = pd.DataFrame(matrix, columns = ['col1','col2','col3'],
                         index = ['row1','row2','row3'])

matrix_df
```

[124]:

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9

[125]:
```
matrix_df['col1']
```

[125]:
```
row1    1
row2    4
row3    7
Name: col1, dtype: int64
```

[126]:
```
matrix_df.loc['row1']
```

[126]:
```
col1    1
col2    2
col3    3
Name: row1, dtype: int64
```

Note that for the rows you need to use `.loc[...]`.

[127]:
```
matrix_df.index
```

[127]:
```
Index(['row1', 'row2', 'row3'], dtype='object')
```

[128]:
```
matrix_df.columns
```

[128]:
```
Index(['col1', 'col2', 'col3'], dtype='object')
```

[129]:
```
col_df = pd.DataFrame([10,11,12], columns = ['col4'],
                      index = ['row1','row2','row3'])
col_df
```

[129]:

	col4
row1	10
row2	11
row3	12

[130]:
```
row_df = col_df.T
row_df
```

[130]:

	row1	row2	row3
col4	10	11	12

Note that .T is the tranpose operation.

[131]:
```
row_df.index = ['row4']
row_df.columns = ['col1','col2','col3']
row_df
```

[131]:

	col1	col2	col3
row4	10	11	12

[132]:
```
matrix_df = pd.concat( [matrix_df, row_df]  )
```

[133]:
```
matrix_df
```

[133]:

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

For a slightly more interesting example, we revist our country codes.

[134]:
```
country_codes = {'country': ['Switzerland', 'France', 'Italy', 'UK', 'Germany'],
                 'code':[41, 33, 39, 44, 49]}

codes_df = pd.DataFrame(country_codes)
codes_df
```

[134]:

	country	code
0	Switzerland	41

```
1       France    33
2        Italy    39
3           UK    44
4      Germany    49
```

[135]:
```
codes_df['country'] == 'UK'
```

[135]:
```
0    False
1    False
2    False
3     True
4    False
Name: country, dtype: bool
```

[136]:
```
codes_df[ codes_df['country'] == 'UK' ]
```

[136]:
```
  country  code
3      UK    44
```

[137]:
```
codes_df.loc[3]
```

[137]:
```
country    UK
code       44
Name: 3, dtype: object
```

1.1.8 Analyse input data and write out a result file

In the following, we read in a table which specifies for different red wines a selection of their respective properties. Each row in the table corresponds to a different wine. We study the data set a little bit. Pandas is well-suited to do data exploration with methods like `groupby` and `describe`.

[6]:
```
import pandas as pd

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')
wine_data
```

[6]:
```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0               7.4             0.700         0.00             1.9      0.076
1               7.8             0.880         0.00             2.6      0.098
2               7.8             0.760         0.04             2.3      0.092
3              11.2             0.280         0.56             1.9      0.075
4               7.4             0.700         0.00             1.9      0.076
…                 …                 …            …               …          …
1594            6.2             0.600         0.08             2.0      0.090
1595            5.9             0.550         0.10             2.2      0.062
1596            6.3             0.510         0.13             2.3      0.076
1597            5.9             0.645         0.12             2.0      0.075
```

```
1598            6.0             0.310        0.47              3.6      0.067

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                    11.0                  34.0  0.99780  3.51       0.56
1                    25.0                  67.0  0.99680  3.20       0.68
2                    15.0                  54.0  0.99700  3.26       0.65
3                    17.0                  60.0  0.99800  3.16       0.58
4                    11.0                  34.0  0.99780  3.51       0.56
…                     …                    …       …      …         …
1594                 32.0                  44.0  0.99490  3.45       0.58
1595                 39.0                  51.0  0.99512  3.52       0.76
1596                 29.0                  40.0  0.99574  3.42       0.75
1597                 32.0                  44.0  0.99547  3.57       0.71
1598                 18.0                  42.0  0.99549  3.39       0.66

      alcohol  quality
0         9.4        5
1         9.8        5
2         9.8        5
3         9.8        6
4         9.4        5
…         …        …
1594     10.5        5
1595     11.2        6
1596     11.0        6
1597     10.2        5
1598     11.0        6

[1599 rows x 12 columns]
```

[140]:
```
wine_data.shape
```

[140]:
```
(1599, 12)
```

[141]:
```
wine_data.head(5)
```

[141]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
```

```
3              17.0                 60.0  0.9980  3.16        0.58
4              11.0                 34.0  0.9978  3.51        0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
```

[142]:
```
wine_data.tail(5)
```

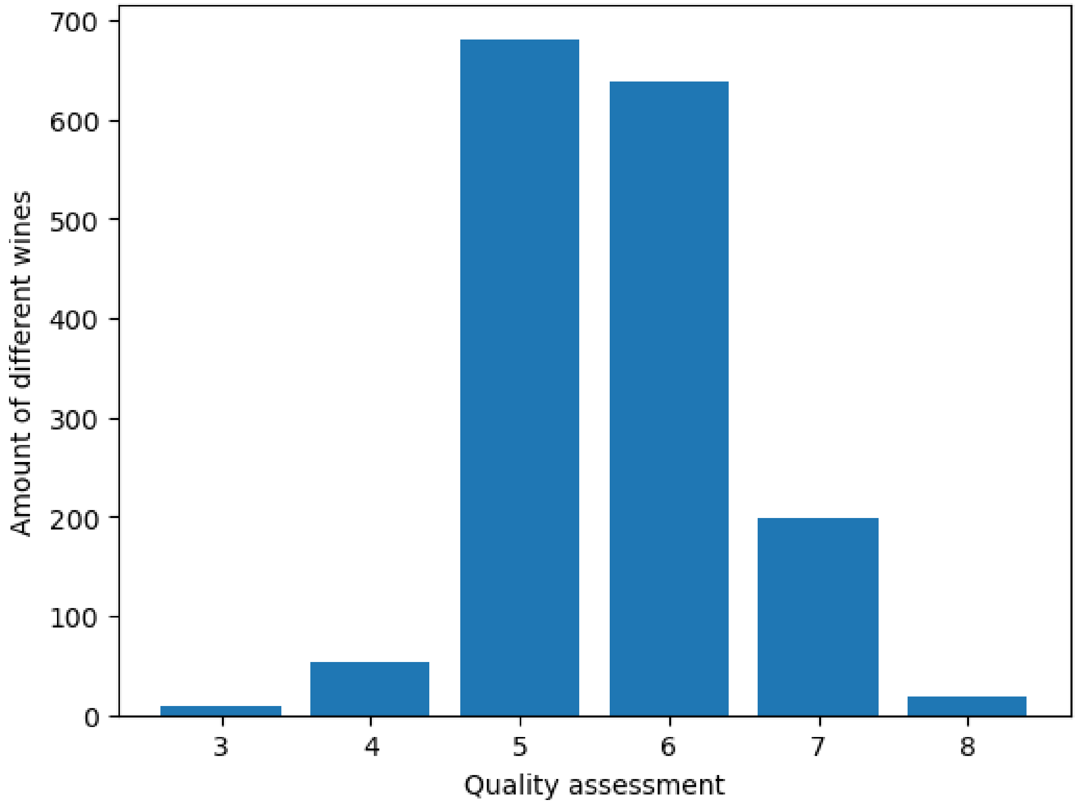
[142]:
```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
1594            6.2             0.600         0.08             2.0      0.090
1595            5.9             0.550         0.10             2.2      0.062
1596            6.3             0.510         0.13             2.3      0.076
1597            5.9             0.645         0.12             2.0      0.075
1598            6.0             0.310         0.47             3.6      0.067

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
1594                 32.0                  44.0  0.99490  3.45       0.58
1595                 39.0                  51.0  0.99512  3.52       0.76
1596                 29.0                  40.0  0.99574  3.42       0.75
1597                 32.0                  44.0  0.99547  3.57       0.71
1598                 18.0                  42.0  0.99549  3.39       0.66

      alcohol  quality
1594     10.5        5
1595     11.2        6
1596     11.0        6
1597     10.2        5
1598     11.0        6
```

Count the number of red wines with a particular quality with `value_counts`.

[143]:
```
quality_counts = wine_data['quality'].value_counts()
print(quality_counts)
```

```
quality
5    681
6    638
7    199
4     53
8     18
3     10
Name: count, dtype: int64
```

[144]:
```
import matplotlib.pyplot as plt

plt.bar(quality_counts.index, quality_counts)
plt.xlabel('Quality assessment')
plt.ylabel('Amount of different wines')
plt.show()
```



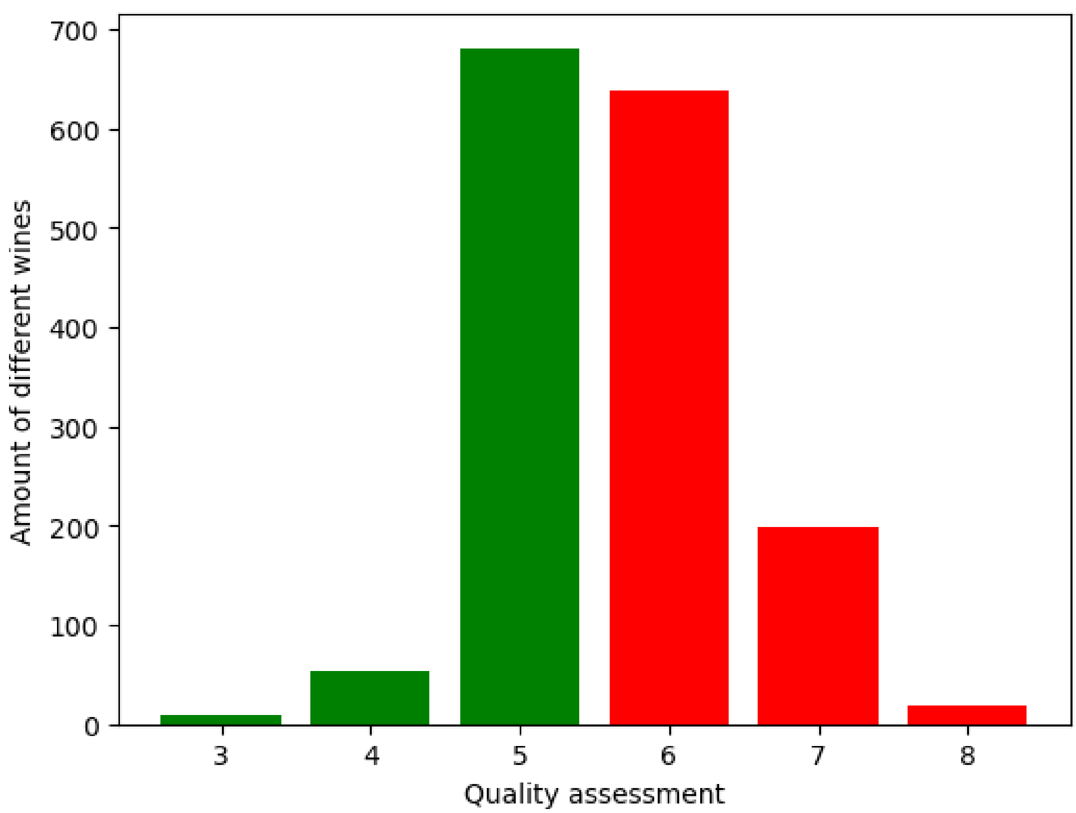
Let us add a new column which classifies whether a red wine is a **premium** wine with a rating larger than 5.

[145]:
```
wine_data['premium'] = wine_data['quality'] > 5

wine_data.head(5)
```

[145]:
```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076
```

```
   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56   
1                 25.0                  67.0   0.9968  3.20       0.68   
2                 15.0                  54.0   0.9970  3.26       0.65   
3                 17.0                  60.0   0.9980  3.16       0.58   
4                 11.0                  34.0   0.9978  3.51       0.56   

   alcohol  quality  premium  
0      9.4        5    False  
1      9.8        5    False  
2      9.8        5    False  
3      9.8        6     True  
4      9.4        5    False  
```

[146]:
```
colours = ['green','red','red','green','red','green']

plt.bar(quality_counts.index, quality_counts, color=colours)
plt.xlabel('Quality assessment')
plt.ylabel('Amount of different wines')
plt.show()
```



Note that in this example the colours where abbreviated. I.e. instead of `color=['green','red','red','green','red','green']` you can use just the initial letter in one string, i.e. `color='grrgrg'`. Also note that the colouring is order by the heights of the bars.

[147]:
```
quality_grouped = wine_data.groupby('quality')
quality_grouped
```

[147]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x137164c10>
```

Note that `groupby('quality')` groups all rows with the same quality together. However, after *collecting* the groups it is a prior not clear how the different rows (with the same quality) are supposed to be combined. Pandas now allows you to choose what operation you would like to perform on the grouped rows. In the following, we see some examples.

For instance, we can start with the actual groups which were identified. A bit similar to dictionaries, the group names are accessed by `.keys()`.

[148]:
```
quality_grouped.groups.keys()
```

[148]:
```
dict_keys([3, 4, 5, 6, 7, 8])
```

Or we can just provide the first row in the respective group with `.first()`.

[149]:
```
quality_grouped.first()
```

[149]:
```
         fixed acidity  volatile acidity  citric acid  residual sugar  \
quality
3                 11.6              0.58         0.66             2.2
4                  7.4              0.59         0.08             4.4
5                  7.4              0.70         0.00             1.9
6                 11.2              0.28         0.56             1.9
7                  7.3              0.65         0.00             1.2
8                  7.9              0.35         0.46             3.6

         chlorides  free sulfur dioxide  total sulfur dioxide  density    pH  \
quality
3            0.074                 10.0                  47.0   1.0008  3.25
4            0.086                  6.0                  29.0   0.9974  3.38
5            0.076                 11.0                  34.0   0.9978  3.51
6            0.075                 17.0                  60.0   0.9980  3.16
7            0.065                 15.0                  21.0   0.9946  3.39
8            0.078                 15.0                  37.0   0.9973  3.35

         sulphates  alcohol  premium
quality
3             0.57      9.0    False
4             0.50      9.0    False
5             0.56      9.4    False
```

```
6            0.58      9.8     True
7            0.47     10.0     True
8            0.86     12.8     True
```

Let's display all rows in group 3 with `.get_group(3)`.

[150]:
```
quality_grouped.get_group(3)
```

[150]:
```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
459            11.6             0.580         0.66            2.20      0.074
517            10.4             0.610         0.49            2.10      0.200
690             7.4             1.185         0.00            4.25      0.097
832            10.4             0.440         0.42            1.50      0.145
899             8.3             1.020         0.02            3.40      0.084
1299            7.6             1.580         0.00            2.10      0.137
1374            6.8             0.815         0.00            1.20      0.267
1469            7.3             0.980         0.05            2.10      0.061
1478            7.1             0.875         0.05            5.70      0.082
1505            6.7             0.760         0.02            1.80      0.078

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
459                  10.0                  47.0  1.00080  3.25       0.57
517                   5.0                  16.0  0.99940  3.16       0.63
690                   5.0                  14.0  0.99660  3.63       0.54
832                  34.0                  48.0  0.99832  3.38       0.86
899                   6.0                  11.0  0.99892  3.48       0.49
1299                  5.0                   9.0  0.99476  3.50       0.40
1374                 16.0                  29.0  0.99471  3.32       0.51
1469                 20.0                  49.0  0.99705  3.31       0.55
1478                  3.0                  14.0  0.99808  3.40       0.52
1505                  6.0                  12.0  0.99600  3.55       0.63

      alcohol  quality  premium
459      9.00        3    False
517      8.40        3    False
690     10.70        3    False
832      9.90        3    False
899     11.00        3    False
1299    10.90        3    False
1374     9.80        3    False
1469     9.70        3    False
1478    10.20        3    False
1505     9.95        3    False
```

[151]:
```
quality_grouped.mean()
```

[151]:
```
         fixed acidity  volatile acidity  citric acid  residual sugar  \
quality
```

```
3             8.360000         0.884500     0.171000         2.635000
4             7.779245         0.693962     0.174151         2.694340
5             8.167254         0.577041     0.243686         2.528855
6             8.347179         0.497484     0.273824         2.477194
7             8.872362         0.403920     0.375176         2.720603
8             8.566667         0.423333     0.391111         2.577778

         chlorides  free sulfur dioxide  total sulfur dioxide   density  \
quality
3         0.122500            11.000000             24.900000  0.997464
4         0.090679            12.264151             36.245283  0.996542
5         0.092736            16.983847             56.513950  0.997104
6         0.084956            15.711599             40.869906  0.996615
7         0.076588            14.045226             35.020101  0.996104
8         0.068444            13.277778             33.444444  0.995212

               pH  sulphates    alcohol  premium
quality
3        3.398000   0.570000   9.955000      0.0
4        3.381509   0.596415  10.265094      0.0
5        3.304949   0.620969   9.899706      0.0
6        3.318072   0.675329  10.629519      1.0
7        3.290754   0.741256  11.465913      1.0
8        3.267222   0.767778  12.094444      1.0
```

There many more methods you can apply to a `groupby` object. A particularly useful one is describe which provides you with some statistics.

[154]:
```
stats = quality_grouped.describe()
stats['alcohol']
```

[154]:
```
         count       mean       std  min     25%     50%     75%   max
quality
3         10.0   9.955000  0.818009  8.4   9.725   9.925  10.575  11.0
4         53.0  10.265094  0.934776  9.0   9.600  10.000  11.000  13.1
5        681.0   9.899706  0.736521  8.5   9.400   9.700  10.200  14.9
6        638.0  10.629519  1.049639  8.4   9.800  10.500  11.300  14.0
7        199.0  11.465913  0.961933  9.2  10.800  11.500  12.100  14.0
8         18.0  12.094444  1.224011  9.8  11.325  12.150  12.875  14.0
```

[155]:
```
stats['sulphates']
```

[155]:
```
         count      mean       std   min     25%    50%    75%   max
quality
3         10.0  0.570000  0.122020  0.40  0.5125  0.545  0.615  0.86
4         53.0  0.596415  0.239391  0.33  0.4900  0.560  0.600  2.00
5        681.0  0.620969  0.171062  0.37  0.5300  0.580  0.660  1.98
6        638.0  0.675329  0.158650  0.40  0.5800  0.640  0.750  1.95
```

```
7         199.0  0.741256  0.135639  0.39  0.6500  0.740  0.830  1.36
8          18.0  0.767778  0.115379  0.63  0.6900  0.740  0.820  1.10
```

[156]:
```
stats['alcohol'].to_csv("data/saved_pandas_example.csv", float_format='%.3f',␣
↪sep = ',', header = True, index = False)
```

Note that similar to `fmt='%1.3f'` for NumPy, `float_format='%.3f'` specifies that the floats shall only have 3 decimals when writte to the .csv file.

1.2 Seaborn with Pandas

Seaborn is a statistical data visualisation library which builds upon matplotlib and uses Pandas data structures. It makes plotting of attractive figures really easy, in particular in combination with Pandas objects.

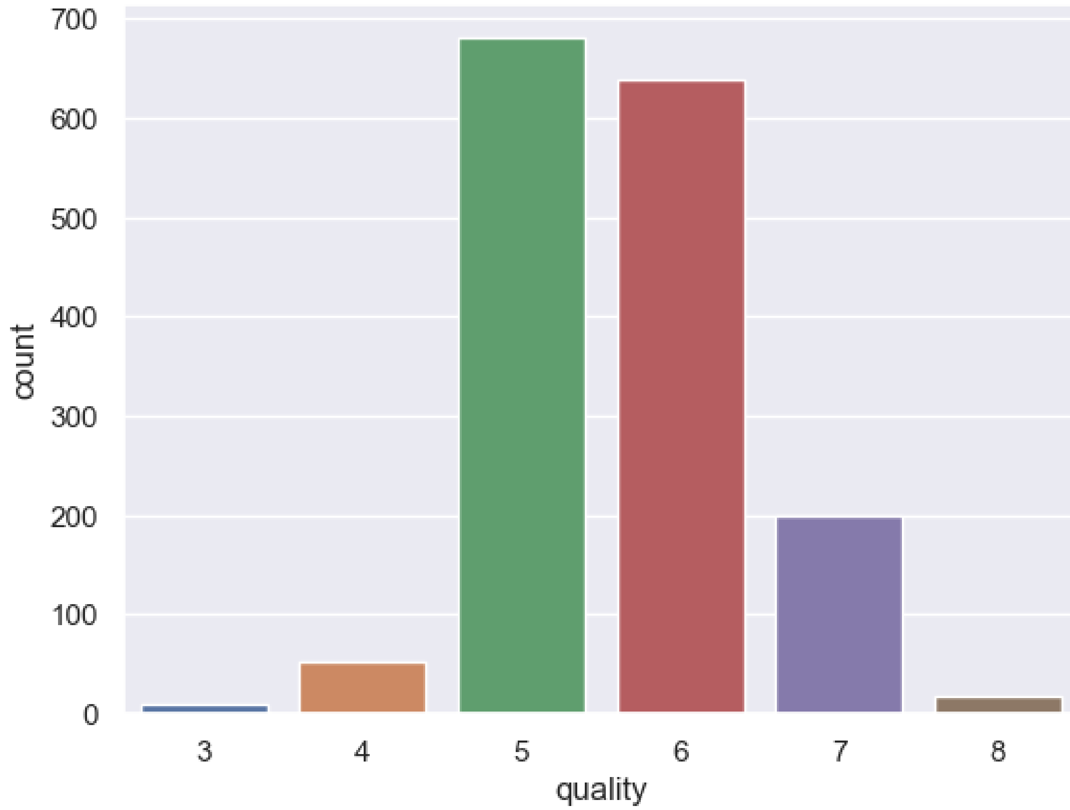
The common abbrevation for seaborn is `sns`.

[2]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

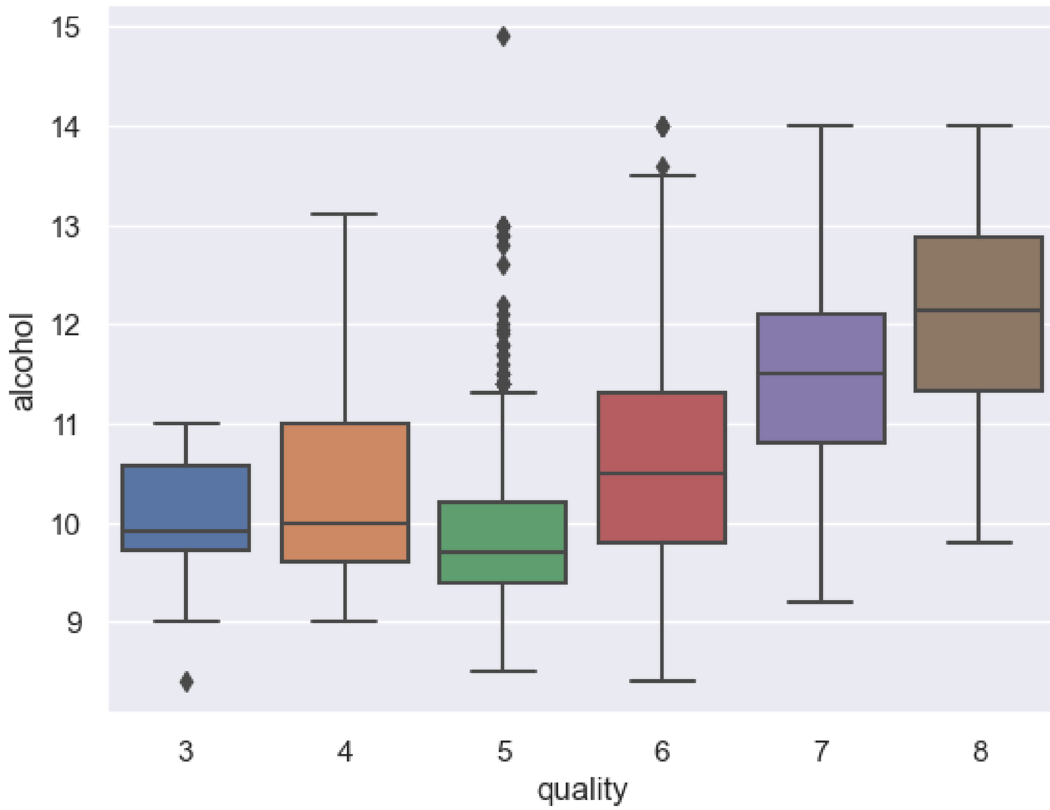
For ggplot stlye plots use:

[3]:
```
sns.set()
```

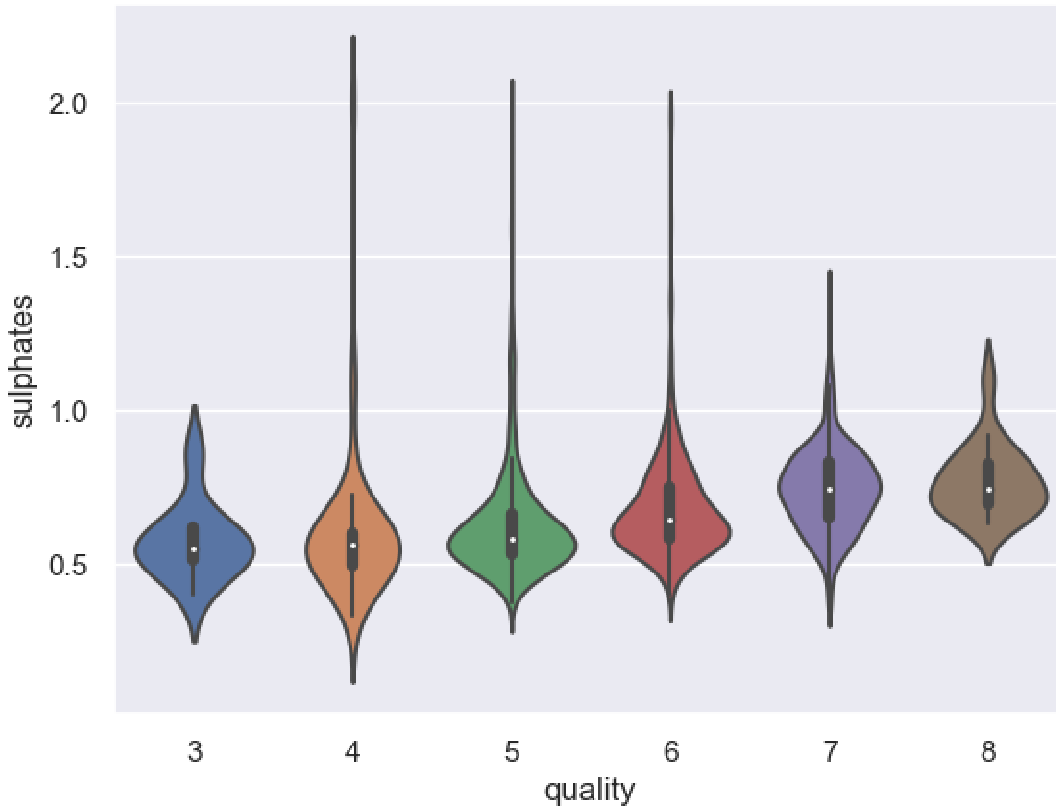
[7]:
```
sns.countplot(x='quality', data=wine_data)
plt.show()
```



[8]:
```
sns.boxplot(x='quality', y='alcohol', data=wine_data)
plt.show()
```



[9]:
```
sns.violinplot(x='quality', y='sulphates', data=wine_data)
plt.show()
```



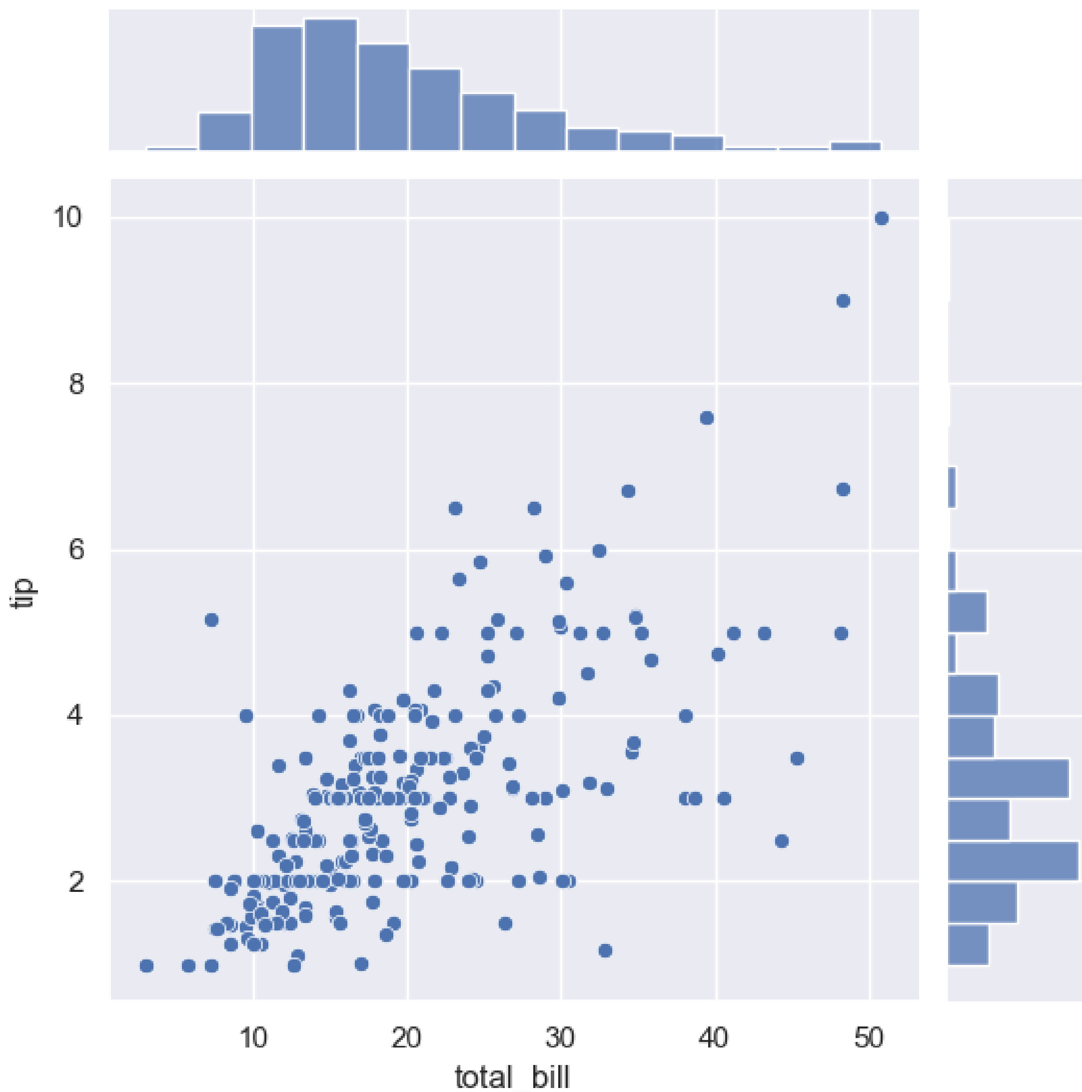
Let's consider one of the standard examples of seaborn, the **tips** data set.

[10]:
```
tips = sns.load_dataset("tips")
tips.head(5)
```

[10]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

[11]:
```
sns.jointplot(x="total_bill", y="tip", data=tips)
plt.show()
```



1.3 Exercise section

(1.) Create a NumPy array with entries from 4 to 9 and reshape the array to have shape (3,2). Make us of `np.arange` and `reshape`. Let's call this matrix `ex1`. Put your solution here:

[ ]:
```

```

Check your result by executing:

[ ]:
```
print(ex1)
```

(2.) Create an array with three random integers between 0 and 20. Make use of `np.random.randint`. Let's call this matrix `rand_ints`. Put your solution here:

[ ]:

```

```

Check your result by executing:

[ ]:

```
print(rand_ints)
```

(3.) Multiply (element-wise) the last column of `ex1` with `rand_ints` and assign the result to the the last column of `ex1`. Put your solution here:

[ ]:

```

```

Check your result by executing:

[ ]:

```
print(ex1)
```

(4.) Append `rand_ints` as a column to matrix `ex1`. Put your solution here:

[ ]:

```

```

Check your result by executing:

[ ]:

```
print(ex1)
```

(5.) Convert NumPy matrix `ex1` into a Pandas dataframe `ex5` and name the columns of `A`, `B` and `C`. Put your solution here:

[ ]:

```

```

Check your result by executing:

[ ]:

```
ex5
```

1.4 Proposed Solutions

(1.) Create a NumPy array with entries from 4 to 9 and reshape the array to have shape (3,2). Make us of `np.arange` and `reshape`. Let's call this matrix `ex1`. Put your solution here:

[12]:

```
ex1 = np.arange(4,10).reshape(3,2)
```

Check your result by executing:

[13]:

```
print(ex1)
```

```
[[4 5]
 [6 7]
 [8 9]]
```

(2.) Create an array with three random integers between 0 and 20. Make use of `np.random.randint`. Let's call this matrix `rand_ints`. Put your solution here:

[14]:

```
rand_ints = np.random.randint(20, size=3)
```

Check your result by executing:

```
[15]: print(rand_ints)
```

```
[14 14  3]
```

(3.) Multiply (element-wise) the last column of `ex1` with `rand_ints` and assign the result to the the last column of `ex1`. Put your solution here:

```
[16]: ex1[:,1] = ex1[:,1]*rand_ints
```

Check your result by executing:

```
[17]: print(ex1)
```

```
[[ 4 70]
 [ 6 98]
 [ 8 27]]
```

(4.) Append `rand_ints` as a column to matrix `ex1`. Put your solution here:

```
[18]: ex1 = np.append(ex1,rand_ints[:,np.newaxis],axis=1)
```

Check your result by executing:

```
[19]: print(ex1)
```

```
[[ 4 70 14]
 [ 6 98 14]
 [ 8 27  3]]
```

(5.) Convert NumPy matrix `ex1` into a Pandas dataframe `ex5` and name the columns of `A`, `B` and `C`. Put your solution here:

```
[20]: ex5 = pd.DataFrame(ex1, columns=['A','B','C'])
```

Check your result by executing:

```
[21]: ex5
```

[21]:

	A	B	C
0	4	70	14
1	6	98	14
2	8	27	3

7-Subroutines_and_OOP

October 4, 2023

1 7. Subroutines and Object-oriented Programming

In the seventh section we learn about object-oriented programming and how to

- define our own functions and
- work with classes and subclasses.

The folder `Notebooks/function_example/` provides example scripts on how you might want to use functions in practice.

Keywords: `def`, `class`, `return`, `help`, `self`

1.1 Functions

Most of the time, there are certain operations which are performed several times in a program. For example, the application of a particular analysis or calculation with different input data, plotting of figures after a new measurement and much more. In the last notebooks we have already encoutered a lot of **built-in functions** like `print` or `type` and other functions like `numpy.mean()`. The basic steps how a function works are:

1. The function is called
2. The function executs some action
3. The function returns some value

Let us consider our own example of a function. The function is defined by a **function signature** and a **function body**. The signature starts with `def` and gives the name of the function, the arguments it expects and ends with a colon `:`. The function body contains the code which is executed when the function gets called. Usually, a `return` statement indicates what will be returned if the function is called. However, it is not necessary to return anything and so `return` can be omitted, too.

[1]:
```
def add_date(a_string, date):
    # Here begins the function body
    dated_string = a_string + '_' + date

    return dated_string
```

[2]:
```
returned_str = add_date(a_string='experiment', date='05-11-19')
returned_str
```

[2]: 'experiment_05-11-19'

[3]:
```
dated_string
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 dated_string

NameError: name 'dated_string' is not defined
```

Note that the variable `dated_string` is only defined in the scope of the function `add_date`. Outside of the function, this variable doesn't exist and cannot be used.

[4]:
```
add_date(a_string='experiment')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 add_date(a_string='experiment')

TypeError: add_date() missing 1 required positional argument: 'date'
```

Note that usually you need to specify all arguments when you call a function. However, you can also specify default values.

[5]:
```
def only_print(another_string, first = '!', last = '?'):
    print(first + another_string + last)
                    # Version 1
    return          # Version 2
    return None     # Version 3
```

[6]:
```
returned_str_2 = only_print(another_string='alpha')
```

```
!alpha?
```

[7]:
```
print(returned_str_2)
```

```
None
```

Note that is different from the case when the variable would not have been defined, like:

[8]:
```
print(returned_str_3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

```
Cell In[8], line 1
----> 1 print(returned_str_3)

NameError: name 'returned_str_3' is not defined
```

[9]:
```
only_print(another_string = 'beta', first = '___', last = '---')
```

```
___beta---
```

[10]:
```
only_print('beta','___','---')
```

```
___beta---
```

[11]:
```
only_print('beta')
```

```
!beta?
```

1.1.1 Convert temperatures

In the following, you can study a more useful function which converts degrees Celsius to degrees Fahrenheit and vice versa.

[13]:
```
def convert_temp(degrees_celsius = None, degrees_fahrenheit = None):
    '''
    This function converts degrees Celsius to degrees Fahrenheit and
    vice versa.

    degree_celsius: Input value in degrees Celsius to be converted to
                    degrees Fahrenheit.
    degree_fahrenheit: Input value in degrees Fahrenheit to be converted
                       to degrees Celsius.
    return: Temperature in the converted units.
    '''

    if degrees_celsius is not None:
        degrees_fahrenheit = degrees_celsius * 9/5 + 32
        print("{} in °C are {} °F".format(degrees_celsius, degrees_fahrenheit))
        return degrees_fahrenheit

    else:
        degrees_celsius = (degrees_fahrenheit - 32) * 5/9
        print("{} in °F are {} °C".format(degrees_fahrenheit, degrees_celsius))
        return degrees_celsius
```

[14]:
```
deg_F = convert_temp(degrees_celsius = 30)
deg_F
```

```
30 in °C are 86.0 °F
```

[14]:
```
86.0
```

[15]:
```
deg_C = convert_temp(degrees_fahrenheit = 23)
deg_C
```

```
23 in °F are -5.0 °C
```

[15]:
```
-5.0
```

[16]:
```
help(convert_temp)
```

```
Help on function convert_temp in module __main__:

convert_temp(degrees_celsius=None, degrees_fahrenheit=None)
    This function converts degrees Celsius to degrees Fahrenheit and
    vice versa.

    degree_celsius: Input value in degrees Celsius to be converted to
                    degrees Fahrenheit.
    degree_fahrenheit: Input value in degrees Fahrenheit to be converted
                       to degrees Celsius.
    return: Temperature in the converted units.
```

[17]:
```
help(list.pop)
```

```
Help on method_descriptor:

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

1.2 Classes and Object-oriented programming

In object-oriented programming (OOP), objects contain information in the form of *attributes* or *properties* and *methods* with which particular operations can be performed. In most OOPs this objects are **instance of classes**. OOP allow modularity and reusability in your code. Let us see what this exactly means in the following.

[18]:
```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def say_hi(self):
        print("{} says: Hi!".format(self.name))
```

You can think of a class as a **blueprint** of an object. Suppose you want to manage different persons with your program. You will need to add different persons which all have similar properties. The `person` class allows us to create many people which have their individual properties.

An “example” or a “realisation” of a class is usually referred to as an **instance** of the class.

[19]:
```
first_person = person('Alice', 30)
```

[20]:
```
first_person.name
```

[20]: 'Alice'

[21]:
```
first_person.age
```

[21]: 30

[22]:
```
first_person.say_hi()
```

```
Alice says: Hi!
```

Note that `say_hi` is a **function** similar to those we defined before. A class function is referred to as a **class method**. More precisely, `say_hi` is an instance method because it requires the instance object `first_person` in order to be callable.

[32]:
```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hi(self):
        print("{} says: Hi!".format(self.name))

    def marries(self, another_person):
        self.spouse = another_person.name
        print('{} marries {}, congratulations!'
              .format(self.name, another_person.name))

    def younger_30(self):
        '''
        Print True if younger than 30.
        '''
        return self.age < 30
```

[24]:
```
first_person = person('Alice', 30)
second_person = person('Bob', 29)
```

[25]:
```
first_person == second_person
```

[25]: False

[26]:
```
first_person.marries(second_person)
second_person.marries(first_person)
```

```
Alice marries Bob, congratulations!
Bob marries Alice, congratulations!
```

[27]:
```
print("{} is married to {}".format(second_person.name, second_person.spouse))
```

```
Bob is married to Alice
```

[28]:
```
print("Did {} celebrate her 30th birthday already? Answer: {}"
      .format(first_person.name, not first_person.younger_30()))
```

```
Did Alice celebrate her 30th birthday already? Answer: True
```

[29]:
```
print("What about {}? Answer: {}"
      .format(second_person.name, not second_person.younger_30()))
```

```
What about Bob? Answer: False
```

[33]:
```
help(person)
```

```
Help on class person in module __main__:

class person(builtins.object)
 |  person(name, age)
 |
 |  Methods defined here:
 |
 |  __init__(self, name, age)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  marries(self, another_person)
 |
 |  say_hi(self)
 |
 |  younger_30(self)
 |      Print True if younger than 30.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
```

```
|      list of weak references to the object (if defined)
```

1.2.1 Subclasses

It is possible to define subclasses which build upon other classes and **inherit** their structure and methods.

[34]:
```python
class child(person):
    def __init__(self, name, age, mother):
        super(child, self).__init__(name, age)
        self.mother = mother

    def print_mother(self):
        print(self.mother.name)
```

[37]:
```python
newborn = child('Charlie', 1, first_person)
```

[36]:
```python
newborn.say_hi()
```

```
Charlie says: Hi!
```

Note that for the class `child` we did not define a method `say_hi`. However, the method was *inherited* from the "parent" class `person`.

[38]:
```python
newborn.print_mother()
```

```
Alice
```

1.3 Exercise section

(1.) In the function `convert_temp` it would be possible to provide both quantities, i.e.

`convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)`.

Currently, the output would be:

[ ]:
```python
convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)
```

However, we would like the function to indicate that this might be not the intended behaviour. Instead we would like to read a message like:

[ ]:
```python
print("You provided the temperature both in degrees Celsius "
      "as well as Fahrenheit. You probably don't need the "
      "conversion, in this case. Otherwise, provide only "
      "one of the two.")
```

Incorporate this behaviour into the function. Make use of the `if` condition and `return`.

[ ]:
```
def convert_temp(degrees_celsius = None, degrees_fahrenheit = None):
    '''...'''
```

Check whether your implementation is correct by executing the following cell:

[ ]:
```
convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)
```

(2.) Add another method `age_in_days(...)` to the `person` class which calculates the age in days and prints the result when called.

[ ]:
```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hi(self):
        print("{} says: Hi!".format(self.name))

    def marries(self, another_person):
        self.spouse = another_person.name
        print('{} and {} just married, congratulations!'
              .format(another_person.name, self.name))

    def younger_30(self):
        return self.age < 30

    '''...'''
```

After adding the new method, create a new person and use the `age_in_days()` method to print the age in days of your created person. Put your solution in the following cell:

[ ]:
```
```

1.4 Proposed Solutions

(1.) In the function `convert_temp` it would be possible to provide both quantities, i.e.

`convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)`.

Currently, the output would be:

[39]:
```
convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)
```

```
0 in °C are 32.0 °F
```

[39]:
```
32.0
```

However, we would like the function to indicate that this might be not the intended behaviour. Instead we would like to read a message like:

[40]:
```
print("You provided the temperature both in degrees Celsius "
      "as well as Fahrenheit. You probably don't need the "
      "conversion, in this case. Otherwise, provide only "
      "one of the two.")
```

```
You provided the temperature both in degrees Celsius as well as Fahrenheit. You
probably don't need the conversion, in this case. Otherwise, provide only one of
the two.
```

Incorporate this behaviour into the function. Make use of the `if` condition and `return`.

[41]:
```
def convert_temp(degrees_celsius = None, degrees_fahrenheit = None):
    '''...'''

    if (degrees_celsius is not None) and (degrees_fahrenheit is not None):
        print("You provided the temperature both in degrees Celsius "
              "as well as Fahrenheit. You probably don't need the "
              "conversion, in this case. Otherwise, provide only "
              "one of the two.")
        return

    elif degrees_celsius is not None:
        degrees_fahrenheit = degrees_celsius * 9/5 + 32
        print("{} in °C are {} °F".format(degrees_celsius, degrees_fahrenheit))

        return degrees_fahrenheit

    else:
        degrees_celsius = (degrees_fahrenheit - 32) * 5/9
        print("{} in °F are {} °C".format(degrees_fahrenheit, degrees_celsius))

        return degrees_celsius
```

Check whether your implementation is correct by executing the following cell:

[42]:
```
convert_temp(degrees_celsius = 0, degrees_fahrenheit = 70)
```

```
You provided the temperature both in degrees Celsius as well as Fahrenheit. You
probably don't need the conversion, in this case. Otherwise, provide only one of
the two.
```

(2.) Add another method to the `person` class which calculates the age in days and prints the result when called.

[43]:
```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hi(self):
```

```
        print("{} says: Hi!".format(self.name))

    def marries(self, another_person):
        self.spouse = another_person.name
        print('{} and {} just married, congratulations!'
              .format(another_person.name, self.name))

    def younger_30(self):
        return self.age < 30

    def age_in_days(self):
        print(self.age*365.25)
```

After adding the new method, create a new person and use the `age_in_days()` method to print the age in days of your created person. Put your solution in the following cell:

[44]:
```
new_person = person('A', 25)
new_person.age_in_days()
```

```
9131.25
```

8-Advanced_Examples

October 4, 2023

1 8. Advanced Examples

In the last section we look at some advanced examples. Here, the idea is that you see how a full script might look like and to get insights what kind of analysis tools / libraries are available.

In the follwing we will have a look at two examples, which use the libraries * `scipy`: SciPy contains modules for scientific computation and builds upon NumPy. * `sklearn`: The scikit-learn library offers a lot efficient implementations for Machine Learning.

Some other interesting libraries you might want to check out:

- `rpy2` for using R in Python: porting code from R to Python
- `wx` for programming Graphical User Interfaces: wxPython overview
- `arcpy` is a geoprocessing tool for combining Python with ArcGIS: ArcPy quick tour
- `sqlite3` for creating and working with SQLite databases: See an example of how to create a table
- `pysam` for reading and manipulating biological sequence data: working with BAM and SAM formatted files

---

2 Hypothesis Testing

Statistical hypothesis testing plays a crucial role in various disciplines. Python and in particular SciPy offers a great variety of already implemented statistical tools. Let's have a look how you can perform a **one sample t-test**.

[1]:
```
from scipy.stats import ttest_1samp
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

First, we create some exemplary data and sample **uniformly** 300 random age values from 0 to 80.

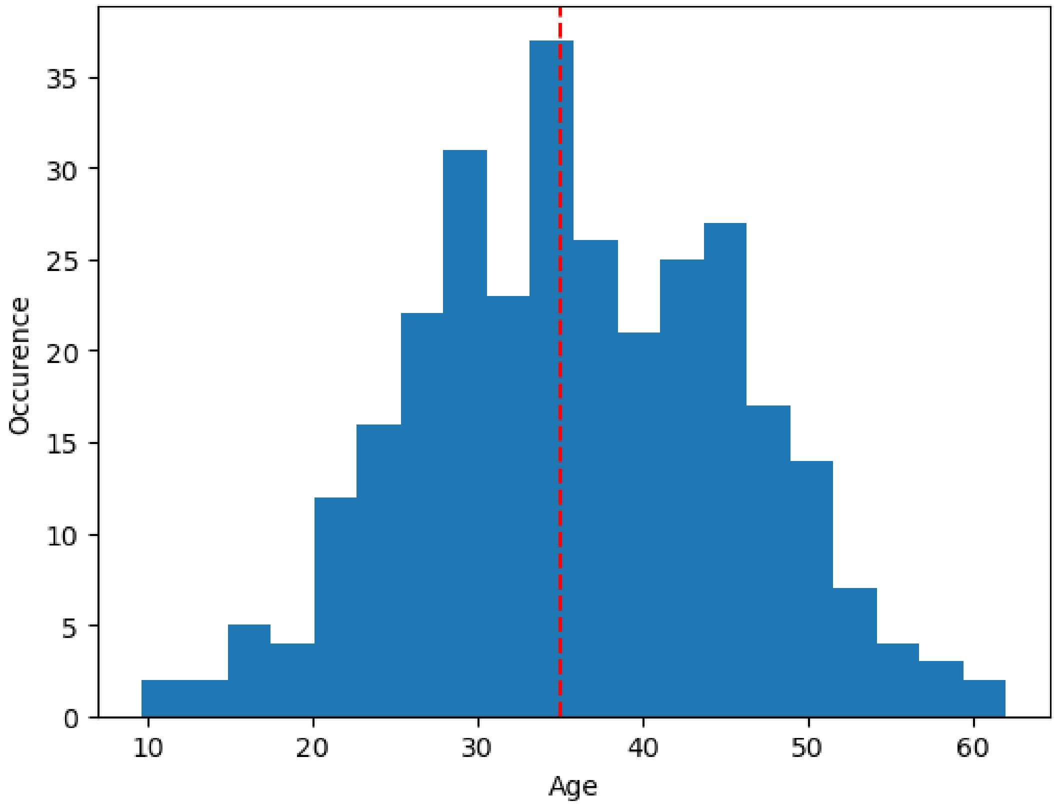
[7]:
```
age = np.random.randint(low=0, high=80, size=300)
```

Or, we could sample from a **normal distribution** with $\mu = 35$ (loc) and $\sigma = 10$ (sacle) 300 random age values.

[11]:
```
age = np.random.normal(loc=35.0, scale=10.0, size=300)
```

We inspect the data with an histrogram where the age values are aggregated in 20 bins.

[12]:
```
plt.hist(age, bins=20)
plt.xlabel('Age')
plt.ylabel('Occurence')
plt.axvline(x=35, color='red', linestyle='--')
plt.show()
```



Let us assume that our **null hypothesis** is that the mean population age is 35, represented by the dashed red line in the plot. The age mean of our data is

[13]:
```
# age_mean = np.mean(age)

age_mean = age.mean()

print('Data age mean:', age_mean, '\n')
```

```
Data age mean: 36.10333355808521
```

We now test the null hypothesis with a one-sample t-test.

[14]:
```
tstat, pval = ttest_1samp(age, popmean=35)

print("p-value:", pval, '\n')

if pval < 0.05:
    print("We reject the null hypothesis")
else:
    print("We accept the null hypothesis")
```

```
p-value: 0.05370340544515489

We accept the null hypothesis
```

---

3 Classification with Random Forests

In the next example, we will try to classify premium red wine from their measured properties (features). Random Forests are very powerful machine learning methods which build on decision trees. A decision tree in our application would look something like this:

Random Forests construct a multitude of such decision trees. The individual decision trees provide a classification result and by majority voting the final classification is obtained.

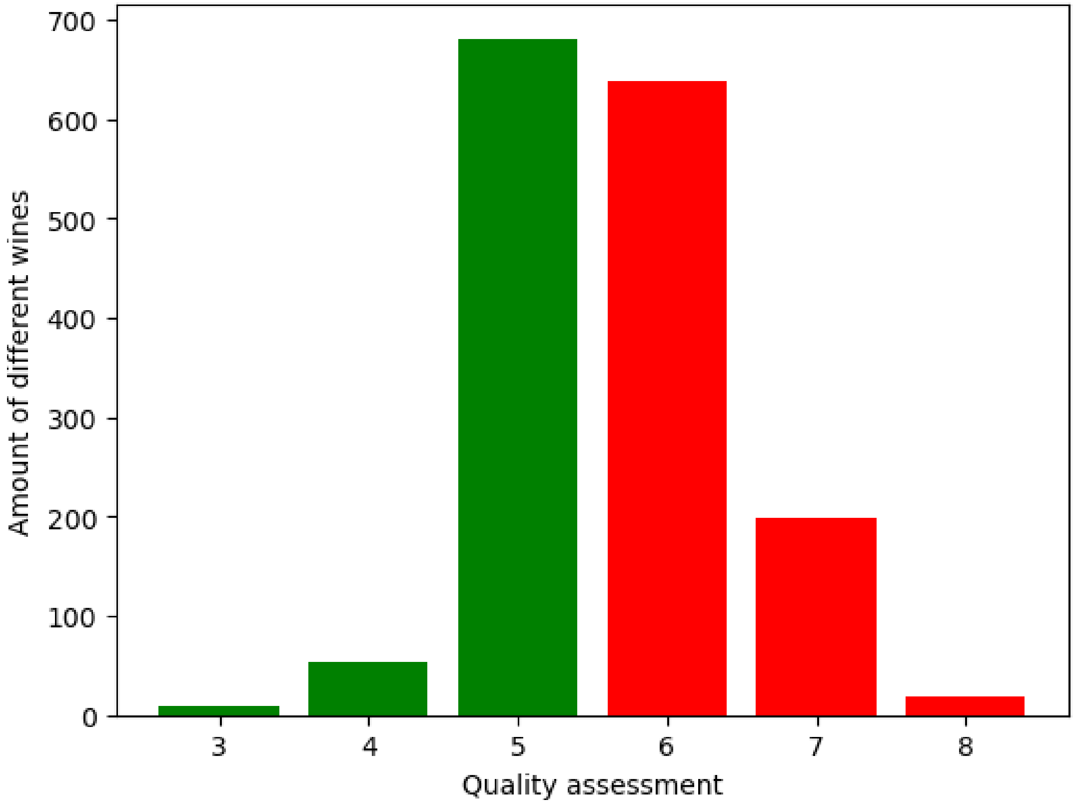
If you want to see a good example of what a “real” class might look like check out the base class for random forests.

[19]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
```

This is what we had before:

[24]:
```
import pandas as pd
import matplotlib.pyplot as plt

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

wine_data['premium'] = wine_data['quality'] > 5
# wine_data['premium'] = wine_data['quality'] > 6

quality_counts = wine_data['quality'].value_counts()

colours = ['green','red','red','green','red','green'] # for > 5
# colours = ['green','green','red','green','red','green'] # for > 6

plt.bar(quality_counts.index, quality_counts, color=colours)
plt.xlabel('Quality assessment')
plt.ylabel('Amount of different wines')
plt.show()
```

```
wine_data.head(5)
```




```
[24]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
      0            7.4              0.70         0.00             1.9      0.076
      1            7.8              0.88         0.00             2.6      0.098
      2            7.8              0.76         0.04             2.3      0.092
      3           11.2              0.28         0.56             1.9      0.075
      4            7.4              0.70         0.00             1.9      0.076

         free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
      0                 11.0                  34.0   0.9978  3.51       0.56
      1                 25.0                  67.0   0.9968  3.20       0.68
      2                 15.0                  54.0   0.9970  3.26       0.65
      3                 17.0                  60.0   0.9980  3.16       0.58
      4                 11.0                  34.0   0.9978  3.51       0.56

         alcohol  quality  premium
      0      9.4        5    False
      1      9.8        5    False
      2      9.8        5    False
```

```
3       9.8        6     True
4       9.4        5    False
```

This is new:

[25]:
```
target = wine_data['premium'].astype(int)
wine_features = wine_data.drop(['quality','premium'], axis=1)

print("Shape of wine_features:\t{}\nShape of target:\t{}\n"
      .format(wine_features.shape, target.shape)
     )
```

```
Shape of wine_features: (1599, 11)
Shape of target:        (1599,)
```

[26]:
```
target.head(5)
```

[26]:
```
0    0
1    0
2    0
3    1
4    0
Name: premium, dtype: int64
```

The Goal is to learn a classifier which predicts whether a wine is a premium / non-premium wine on the basis of the measured wine features.

Select (randomly) a set on which the Random Forest classifier is calibrated / trained on and a test set on which the performance is assessed. We consider a test set size of 30% of the original data set.

[27]:
```
feat_train, feat_test, target_train, target_test = train_test_split(
    wine_features, target, test_size = 0.3)

print("After splitting into train and test sets:\n\n"
      "Shape of feat_train:\t{}\nShape of target_train:\t{}\n"
      "Shape of feat_test:\t{}\nShape of target_test:\t{}"
      .format(feat_train.shape, target_train.shape, feat_test.shape,
              target_test.shape)
     )
```

```
After splitting into train and test sets:

Shape of feat_train:    (1119, 11)
Shape of target_train:  (1119,)
Shape of feat_test:     (480, 11)
Shape of target_test:   (480,)
```

[28]:
```
random_forest = RandomForestClassifier(n_estimators=100, max_depth=8)
random_forest.fit(feat_train, target_train)
```

[28]: RandomForestClassifier(max_depth=8)

[29]:
```
correct_pred = random_forest.predict(feat_test) == target_test

correct = correct_pred.value_counts()

accuracy = (correct[True] / (correct[True] + correct[False]))*100
print("The random forest identified premium / non-premium wines with {}%␣
 ↪accuracy!"
      .format(accuracy))
```

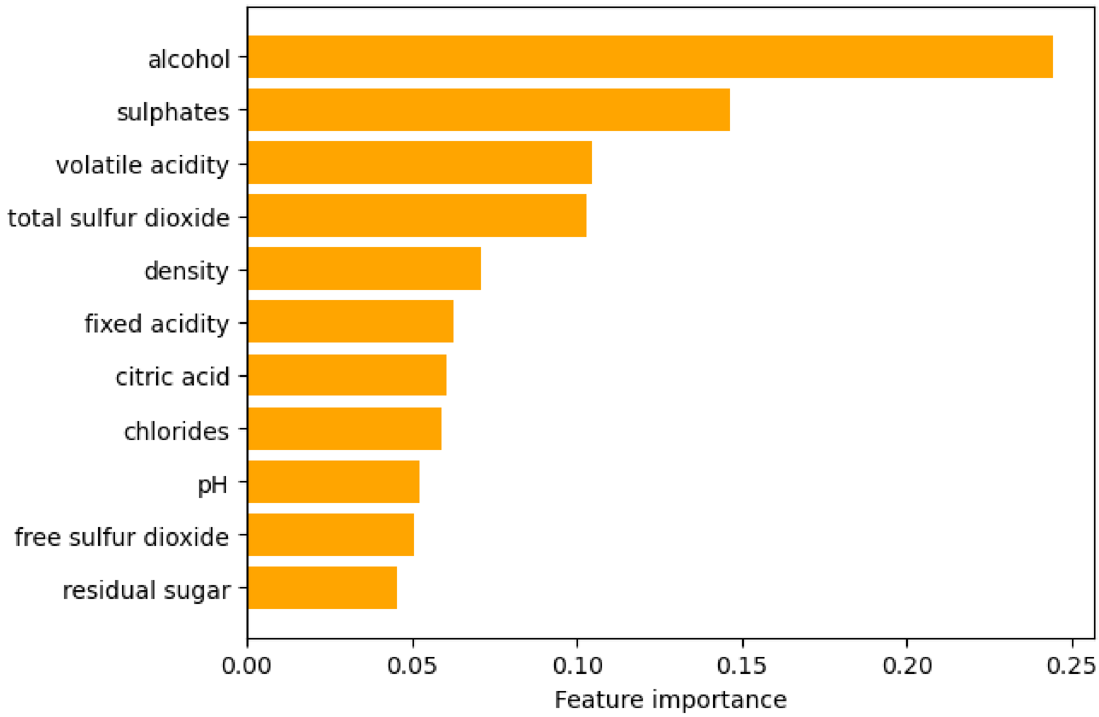
```
The random forest identified premium / non-premium wines with 81.25% accuracy!
```

Let's check what is actually predicted wrongly:

[30]:
```
rel_incorrect_pred = target_test[correct_pred == False].value_counts() /␣
 ↪target_test.value_counts()

print("Incorrect predictions by premium quality:\n{}".
 ↪format(rel_incorrect_pred))
print("\nRatio of premium / non-premium wines in test set:\n{}"
      .format(target_test.value_counts(normalize=True))
     )
```

```
Incorrect predictions by premium quality:
premium
1    0.184906
0    0.190698
Name: count, dtype: float64

Ratio of premium / non-premium wines in test set:
premium
1    0.552083
0    0.447917
Name: proportion, dtype: float64
```

[31]:
```
feature_scores = random_forest.feature_importances_
feature_names = list(wine_features.columns)

important_features = pd.Series(feature_scores, index=feature_names).
 ↪sort_values()

plt.barh(important_features.index, important_features, color='Orange')
plt.xlabel('Feature importance')
plt.show()
```


alcohol
sulphates
volatile acidity
total sulfur dioxide
density
fixed acidity
citric acid
chlorides
pH
free sulfur dioxide
residual sugar
0.00
0.05
0.10
0.15
0.20
0.25
Feature importance

9-Exercises

October 4, 2023

1 9. Exercises

This notebook provides three additional blocks of exercises covering different areas. You can choose the topics you are more interested in and start with the corresponding block.

1. **Elevations in Switzerland**: We make use of **NumPy** and **Matplotlib** to analyse the elevation profile of Switzerland.
2. **Printing Patterns**: A slighlty more creative use of **loops** and **conditional statements** is considered to print particular patterns.
3. **Analysing the Tips Dataset**: We have a look at the tips dataset again and work with **Pandas** and **Seaborn**.

1.1 1. Elevations in Switzerland

In the following, you are provided with a NumPy array (stored as a `.npy` file in `data`) containing data on the elevations of Switzerland, normalised to the range [0,1], and visualised below. We will make use of **NumPy** and **Matplotlib** to analyse this dataset a little bit further.

The next cell loads the required modules, the dataset, and the maximum elevation (in meters) to rescale the normalised values.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

max_value = 4632.51

ch_tiled = np.load("data/switzerland.npy")
```

(1.1) Inspect array Inspect the array shape (put solution in the next cell)

[ ]:
```

```

and an exemplary tile, e.g. tile 4 with index 3 (put solution in the next cell)

[ ]:
```

```

(1.2) Rescale values Undo the normalisation to the [0,1] range by using `max_value` and store the result in `ch_tiled`.

Put your solution here:

```
[ ]:
```

(1.3) Identify tile with highest elevation Firstly, indentify the maximum value in each tile and store the result in `max_per_tile` such that the output looks like

```
In: print(max_per_tile)
Out: [          nan           nan           nan 1816.67058824  944.66870588
           nan           nan 2125.50458824 2270.83823529 2234.50482353
 2706.83917647           nan 2252.67152941 2579.67223529 2924.83964706
 3560.67435294 3742.34141176 3905.84176471 2634.17235294 3215.50694118
 4360.00941176 4033.00870588 3905.84176471 3905.84176471 2652.33905882
 3815.00823529 4505.34305882 3851.34164706 3887.67505882 4251.00917647
           nan 4396.34282353 4596.17658824 2034.67105882 2688.67247059
           nan]
```

Secondly, identify the tile with the largest value, i.e. the highest elevation.

Hints: * Make use of `np.nanmax` and `np.nanargmax` which ignore `nan` values to find the maximum value. * Check out the effect of `axis=0` and `axis=(1,2)` when identifying the maximum value.

Put your solution for `max_per_tile` here:

```
[ ]:
```

Put your solution for `max_tile_index` here:

```
[ ]:
```

(1.4) Which tiles are all empty? Identify the tiles which contain only background, i.e. `nan`.

Hint: Make use of the functions `np.all` and `np.isnan`. You might want to check out the following documentation:

```
[ ]: help(np.all)
```

```
[ ]: help(np.isnan)
```

Put your solution here:

```
[ ]:
```

(1.5) Flatten array into vector Flatten `ch_tiled` into a vector and store the resulting vector in `ch_flat`.

Put your solution here:

```
[ ]:
```

(1.6) Plot histogram of the different elevation levels Plot a histogram by making use of `ch_flat` and use 50 bins.

Put your solution here:

```
[ ]:
```

Bonus exercise (1.7): Recreate the elevation plot As a bonus exercise, try to recreate the plot shown in the beginning of the exercise as much as possible.

Hints: Make use of `fig, ax = plt.subplots(...)` and `.imshow(tile, vmin=0, vmax=max_value)`

You can put your solution here:

```
[ ]:
```

1.2 2. Printing Patterns

In this exercise, we review **loops** and **conditional statements** and make a slightly more creative use of them to print the following patterns.

General hint: You can use `range` in descending order like this

```
for i in range(5,0,-1):
    print(i)
```

```
Out:
5
4
3
2
1
```

(2.1) Print pattern Recreate the following pattern:

```
*
**
***
****
*****
****
***
**
*

#
##
###
####
```

```
#####
####
###
##
#

x
xx
xxx
xxxx
xxxxx
xxxx
xxx
xx
x
```

Use

```
[ ]: symbol = ['*','#','x']
     rows = 5
```

and put your solution here:

```
[ ]:
```

(2.2) Print pattern Recreate the following pattern:

```
*
**
***
****
xxxxx
****
***
**
*
```

Put your solution here:

```
[ ]:
```

(2.3) Print pattern Recreate the following pattern:

```
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
```

Hint: You can use the argument `end=' '` in the `print` function in the following way

```
for i in range(5):
    print(i, end=' ')
```

```
Out: 0 1 2 3 4
```

In other words, line breaks are replaced by spaces, because

```
for i in range(5):
    print(i)
```

```
Out:
0
1
2
3
4
```

Put your solution here:

```
[ ]:
```

1.3 3. Analysing the Tips Dataset

Here, we reconsider the `tips` dataset encountered in the `6-Numpy_Pandas` notebook and work with **Pandas** and **Seaborn**.

Each row corresponds to an individual visit at a restaurant, with indication of the **day** and **time** of the visit and **size** of the group, whether there were any **smoker** in the group and the **total_bill** and **tip** in dollars as well as the **sex** of the person who payed.

```
[ ]: import pandas as pd
     import seaborn as sns
     import matplotlib.pyplot as plt
```

```
[ ]: tips = sns.load_dataset("tips")
     tips.head(5)
```

(3.1) Dataset overview Make use of `describe()` to get an overview of the dataset

```
[ ]:
```

and check out the additional argument `include=['category']`

```
[ ]:
```

(3.2) Unique Categories Identify for each column containing a categorical variable the unique categories. You might want to make use of `.unique()`.

```
[ ]:
```

(3.3) Mean values by groups Group by categorical variables and obtain the mean values for the numerical variables. E.g. group by `'day'` and obtain the mean values for `'total_bill'`, `'tip'`, `'size'`.

[ ]:

(3.4) Pairplot Try to create a pairplot (check out the documentation in the link) where you differentiate between male and female, i.e. use `'sex'` to plot these aspecets in two different colours.

[ ]:

(3.5) Category Plots Try to create a catplot (check out the documentation in the link), which plots the `'size'` on the x-axis and the **count** of different group sizes on the y-axis (i.e. a histogram) and differentiate between male and female, i.e. use `'sex'` to plot these aspecets in two different colours.

[ ]:

Now, try to create a catplot (check out the documentation in the link), which plots the `'day'` on the x-axis and the **count** of different group sizes on the y-axis (i.e. a histogram). This time create two subplots, one for female and one for male, i.e. use `'sex'` for the plot **columns**.

[ ]:

(3.6) Correlation In this last exercise, we compute the correlation between the numerical variables. To this end, use the method `corr()` and put your solution in the following cell:

[ ]:

Try to create a regplot (check out the documentation in the link), which plots the `'total_bill'` on the x-axis and the `'tip'` on the y-axis.

[ ]:

9-Solutions

October 4, 2023

1 9. Exercises

This notebook provides three additional blocks of exercises covering different areas. You can choose the topics you are more interested in and start with the corresponding block.

1. **Elevations in Switzerland**: We make use of **NumPy** and **Matplotlib** to analyse the elevation profile of Switzerland.
2. **Printing Patterns**: A slighlty more creative use of **loops** and **conditional statements** is considered to print particular patterns.
3. **Analysing the Tips Dataset**: We have a look at the tips dataset again and work with **Pandas** and **Seaborn**.

1.1 1. Elevations in Switzerland

In the following, you are provided with a NumPy array (stored as a `.npy` file in `data`) containing data on the elevations of Switzerland, normalised to the range [0,1], and visualised below. We will make use of **NumPy** and **Matplotlib** to analyse this dataset a little bit further.

The next cell loads the required modules, the dataset, and the maximum elevation (in meters) to rescale the normalised values.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

max_value = 4632.51

ch_tiled = np.load("data/switzerland.npy")
```

(1.1) Inspect array Inspect the array shape (put solution in the next cell)

[2]:
```
print("Shape of ch_tiled:", ch_tiled.shape)
```

```
Shape of ch_tiled: (36, 150, 200)
```

and an exemplary tile, e.g. tile 4 with index 3 (put solution in the next cell)

[3]:
```
print(ch_tiled[3])
```

```
[[        nan        nan        nan …        nan        nan        nan]
 [        nan        nan        nan …        nan        nan        nan]
 [        nan        nan        nan …        nan        nan        nan]
 …
 [        nan        nan        nan … 0.20392157 0.20392157 0.20392157]
 [        nan        nan        nan … 0.20392157 0.20392157 0.20392157]
 [        nan        nan        nan … 0.20392157 0.20392157 0.20392157]]
```

(1.2) Rescale values Undo the normalisation to the [0,1] range by using `max_value` and store the result in `ch_tiled`.

Put your solution here:

[4]:
```
ch_tiled = ch_tiled * max_value
```

(1.3) Identify tile with highest elevation Firstly, indentify the maximum value in each tile and store the result in `max_per_tile` such that the output looks like

```
In: print(max_per_tile)
Out: [          nan           nan           nan 1816.67058824  944.66870588
          nan           nan 2125.50458824 2270.83823529 2234.50482353
 2706.83917647           nan 2252.67152941 2579.67223529 2924.83964706
 3560.67435294 3742.34141176 3905.84176471 2634.17235294 3215.50694118
 4360.00941176 4033.00870588 3905.84176471 3905.84176471 2652.33905882
 3815.00823529 4505.34305882 3851.34164706 3887.67505882 4251.00917647
          nan 4396.34282353 4596.17658824 2034.67105882 2688.67247059
          nan]
```

Secondly, identify the tile with the largest value, i.e. the highest elevation.

Hints: * Make use of `np.nanmax` and `np.nanargmax` which ignore `nan` values to find the maximum value. * Check out the effect of `axis=0` and `axis=(1,2)` when identifying the maximum value.

Put your solution for `max_per_tile` here:

[5]:
```
max_per_tile = np.nanmax(ch_tiled, axis=(1,2))
print(max_per_tile)
```

```
[          nan           nan           nan 1816.67058824  944.66870588
          nan           nan 2125.50458824 2270.83823529 2234.50482353
 2706.83917647           nan 2252.67152941 2579.67223529 2924.83964706
 3560.67435294 3742.34141176 3905.84176471 2634.17235294 3215.50694118
 4360.00941176 4033.00870588 3905.84176471 3905.84176471 2652.33905882
 3815.00823529 4505.34305882 3851.34164706 3887.67505882 4251.00917647
          nan 4396.34282353 4596.17658824 2034.67105882 2688.67247059
          nan]
```

```
/var/folders/y_/2qg_gjq937z8c0xctbn85x8r0000gn/T/ipykernel_6728/3432440464.py:1:
RuntimeWarning: All-NaN slice encountered
  max_per_tile = np.nanmax(ch_tiled, axis=(1,2))
```

Put your solution for `max_tile_index` here:

[6]:
```
max_tile_index = np.nanargmax(max_per_tile)
print(max_tile_index)
```

```
32
```

(1.4) Which tiles are all empty? Identify the tiles which contain only background, i.e. `nan`.

Hint: Make use of the functions `np.all` and `np.isnan`. You might want to check out the following documentation:

[7]:
```
help(np.all)
```

```
Help on _ArrayFunctionDispatcher in module numpy:

all(a, axis=None, out=None, keepdims=<no value>, *, where=<no value>)
    Test whether all array elements along a given axis evaluate to True.

    Parameters
    ----------
    a : array_like
        Input array or object that can be converted to an array.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a logical AND reduction is performed.
        The default (``axis=None``) is to perform a logical AND over all
        the dimensions of the input array. `axis` may be negative, in
        which case it counts from the last to the first axis.

        .. versionadded:: 1.7.0

        If this is a tuple of ints, a reduction is performed on multiple
        axes, instead of a single axis or all the axes as before.
    out : ndarray, optional
        Alternate output array in which to place the result.
        It must have the same shape as the expected output and its
        type is preserved (e.g., if ``dtype(out)`` is float, the result
        will consist of 0.0's and 1.0's). See :ref:`ufuncs-output-type` for more
        details.

    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `all` method of sub-classes of
        `ndarray`, however any non-default value will be.  If the
        sub-class' method does not implement `keepdims` any
```

```
    exceptions will be raised.

where : array_like of bool, optional
    Elements to include in checking for all `True` values.
    See `~numpy.ufunc.reduce` for details.

    .. versionadded:: 1.20.0

Returns
-------
all : ndarray, bool
    A new boolean or array is returned unless `out` is specified,
    in which case a reference to `out` is returned.

See Also
--------
ndarray.all : equivalent method

any : Test whether any element along a given axis evaluates to True.

Notes
-----
Not a Number (NaN), positive infinity and negative infinity
evaluate to `True` because these are not equal to zero.

Examples
--------
>>> np.all([[True,False],[True,True]])
False

>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False])

>>> np.all([-1, 4, 5])
True

>>> np.all([1.0, np.nan])
True

>>> np.all([[True, True], [False, True]], where=[[True], [False]])
True

>>> o=np.array(False)
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array(True)) # may vary
```

[8]:
```
help(np.isnan)
```

```
Help on ufunc:

isnan = <ufunc 'isnan'>
    isnan(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])

    Test element-wise for NaN and return result as a boolean array.

    Parameters
    ----------
    x : array_like
        Input array.
    out : ndarray, None, or tuple of ndarray and None, optional
        A location into which the result is stored. If provided, it must have
        a shape that the inputs broadcast to. If not provided or None,
        a freshly-allocated array is returned. A tuple (possible only as a
        keyword argument) must have length equal to the number of outputs.
    where : array_like, optional
        This condition is broadcast over the input. At locations where the
        condition is True, the `out` array will be set to the ufunc result.
        Elsewhere, the `out` array will retain its original value.
        Note that if an uninitialized `out` array is created via the default
        ``out=None``, locations within it where the condition is False will
        remain uninitialized.
    **kwargs
        For other keyword-only arguments, see the
        :ref:`ufunc docs <ufuncs.kwargs>`.

    Returns
    -------
    y : ndarray or bool
        True where ``x`` is NaN, false otherwise.
        This is a scalar if `x` is a scalar.

    See Also
    --------
    isinf, isneginf, isposinf, isfinite, isnat

    Notes
    -----
    NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
    (IEEE 754). This means that Not a Number is not equivalent to infinity.

    Examples
    --------
    >>> np.isnan(np.nan)
```

```
True
>>> np.isnan(np.inf)
False
>>> np.isnan([np.log(-1.),1.,np.log(0)])
array([ True, False, False])
```

Put your solution here:

[9]:
```python
empty_tiles = np.all(np.isnan(ch_tiled), axis=(1,2))
print(empty_tiles)
```

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

(1.5) Flatten array into vector Flatten `ch_tiled` into a vector and store the resulting vector in `ch_flat`.

Put your solution here:

[10]:
```python
ch_tiled.shape
```

[10]:
```
(36, 150, 200)
```

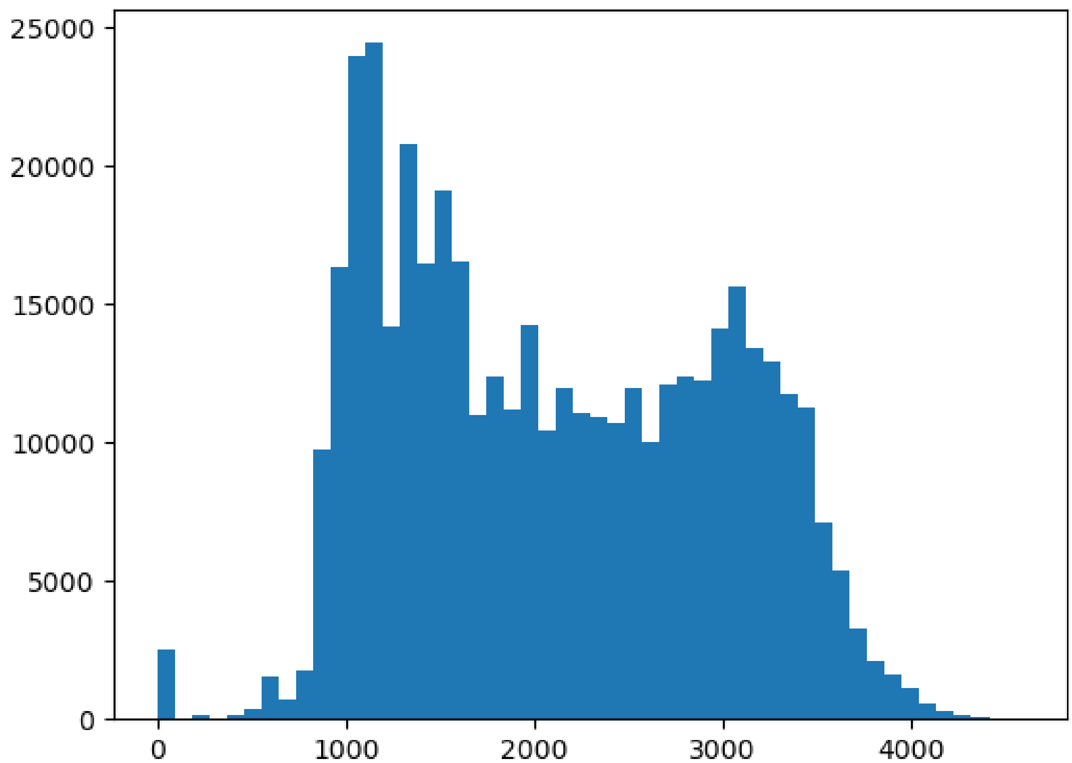
[11]:
```python
ch_flat = ch_tiled.reshape(-1)
print(ch_flat.shape)
```

```
(1080000,)
```

(1.6) Plot histogram of the different elevation levels Plot a histogram by making use of `ch_flat` and use 50 bins.

Put your solution here:

[12]:
```python
plt.hist(ch_flat, bins=50)
plt.show()
```

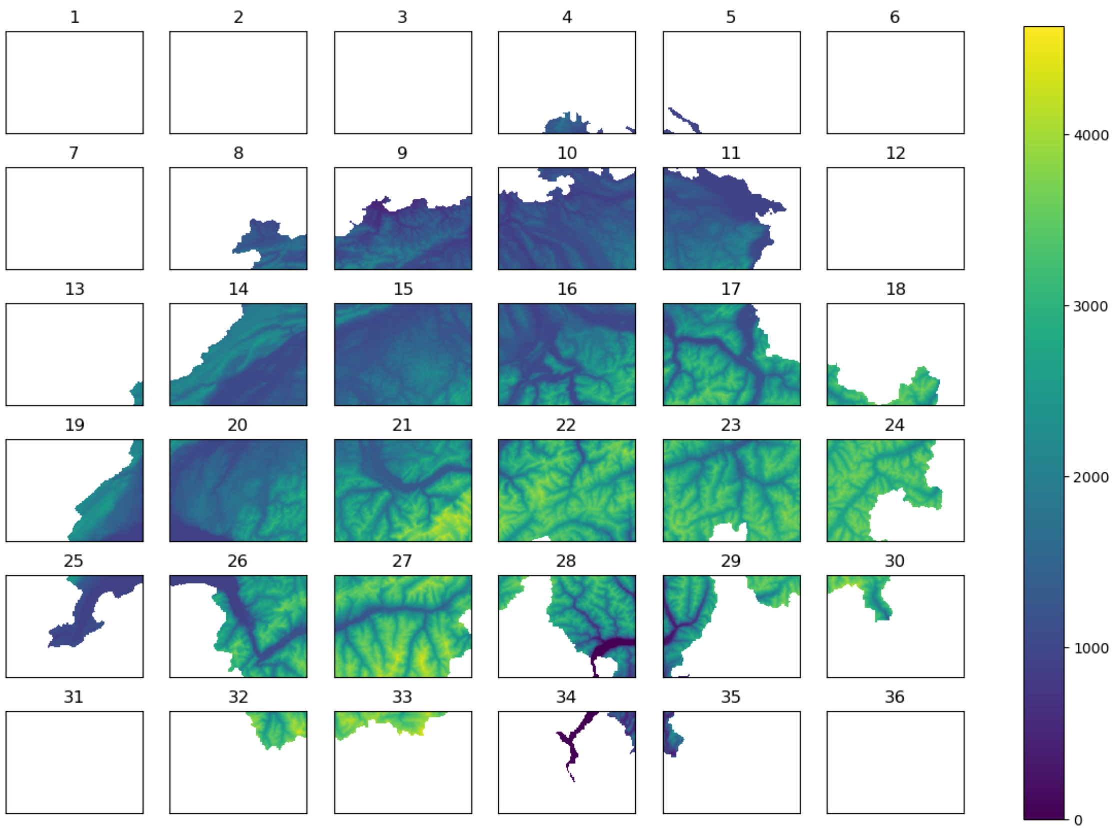


Bonus exercise (1.7): Recreate the elevation plot As a bonus exercise, try to recreate the plot shown in the beginning of the exercise as much as possible.

Hints: Make use of `fig, ax = plt.subplots(...)` and `.imshow(tile, vmin=0, vmax=max_value)`

You can put your solution here:

[13]:
```
fig, ax = plt.subplots(6, 6, figsize=(15, 10))
ax_flat = ax.flat

for i, tile in enumerate(ch_tiled):
    im = ax_flat[i].imshow(tile, vmin=0, vmax=max_value)

    ax_flat[i].axes.get_xaxis().set_visible(False)
    ax_flat[i].axes.get_yaxis().set_visible(False)
    ax_flat[i].set_title(i+1)

fig.colorbar(im, ax=list(ax_flat))

plt.show()
```



1.2 2. Printing Patterns

In this exercise, we review **loops** and **conditional statements** and make a slightly more creative use of them to print the following patterns.

General hint: You can use `range` in descending order like this

```
for i in range(5,0,-1):
    print(i)
```

```
Out:
5
4
3
2
1
```

(2.1) Print pattern Recreate the following pattern:

```
*
**
```

```
***
****
*****
****
***
**
*

#
##
###
####
#####
####
###
##
#

x
xx
xxx
xxxx
xxxxx
xxxx
xxx
xx
x
```

Use

[14]:
```
symbol = ['*','#','x']
rows = 5
```

and put your solution here:

[15]:
```
for s in symbol:
    for i in range(0, rows):
        print(s*i)

    for i in range(rows, 0, -1):
        print(s*i)
```

```

*
**
***
****
*****
****
```

```
***
**
*

#
##
###
####
#####
####
###
##
#

x
xx
xxx
xxxx
xxxxx
xxxx
xxx
xx
x
```

(2.2) Print pattern Recreate the following pattern:

```
*
**
***
****
xxxxx
****
***
**
*
```

Put your solution here:

[16]:
```
for i in range(0, rows):
    print('*'*i)

for i in range(rows, 0, -1):
    if i == rows:
        print('x'*i)
    else:
        print('*'*i)
```

```
*
```

```
**
***
****
xxxxx
****
***
**
*
```

(2.3) Print pattern Recreate the following pattern:

```
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
```

Hint: You can use the argument `end=' '` in the `print` function in the following way

```
for i in range(5):
    print(i, end=' ')
```

```
Out: 0 1 2 3 4
```

In other words, line breaks are replaced by spaces, because

```
for i in range(5):
    print(i)
```

```
Out:
0
1
2
3
4
```

Put your solution here:

[17]:
```
for i in range(rows):
    for j in range(rows):
        if j <= i:
            print(i, end=' ')
        else:
            print(j, end=' ')
    print()
```

```
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
```

1.3 3. Analysing the Tips Dataset

Here, we reconsider the `tips` dataset encountered in the `6-Numpy_Pandas` notebook and work with **Pandas** and **Seaborn**.

Each row corresponds to an individual visit at a restaurant, with indication of the **day** and **time** of the visit and **size** of the group, whether there were any **smoker** in the group and the **total_bill** and **tip** in dollars as well as the **sex** of the person who payed.

[18]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[19]:
```
tips = sns.load_dataset("tips")
tips.head(5)
```

[19]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

(3.1) Dataset overview Make use of `describe()` to get an overview of the dataset

[20]:
```
tips.describe()
```

[20]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

and check out the additional argument `include=['category']`

[21]:
```
tips.describe(include=['category'])
```

[21]:

	sex	smoker	day	time
count	244	244	244	244
unique	2	2	4	2
top	Male	No	Sat	Dinner
freq	157	151	87	176

(3.2) Unique Categories Identify for each column containing a categorical variable the unique categories. You might want to make use of `.unique()`.

[22]:
```
tips['sex'].unique()
```

[22]:
```
['Female', 'Male']
Categories (2, object): ['Male', 'Female']
```

[23]:
```
tips['smoker'].unique()
```

[23]:
```
['No', 'Yes']
Categories (2, object): ['Yes', 'No']
```

[24]:
```
tips['day'].unique()
```

[24]:
```
['Sun', 'Sat', 'Thur', 'Fri']
Categories (4, object): ['Thur', 'Fri', 'Sat', 'Sun']
```

[25]:
```
tips['time'].unique()
```

[25]:
```
['Dinner', 'Lunch']
Categories (2, object): ['Lunch', 'Dinner']
```

(3.3) Mean values by groups Group by categorical variables and obtain the mean values for the numerical variables. E.g. group by `'day'` and obtain the mean values for `'total_bill'`, `'tip'`, `'size'`.

[26]:
```
tips.groupby('sex').mean(numeric_only=True)
```

[26]:

sex	total_bill	tip	size
Male	20.744076	3.089618	2.630573
Female	18.056897	2.833448	2.459770

[27]:
```
tips.groupby('day').mean(numeric_only=True)
```

[27]:

day	total_bill	tip	size
Thur	17.682742	2.771452	2.451613
Fri	17.151579	2.734737	2.105263
Sat	20.441379	2.993103	2.517241
Sun	21.410000	3.255132	2.842105

[28]:
```
tips.groupby('time').mean(numeric_only=True)
```

[28]:

time	total_bill	tip	size
Lunch	17.168676	2.728088	2.411765
Dinner	20.797159	3.102670	2.630682

[29]:
```
tips.groupby('smoker').mean(numeric_only=True)
```

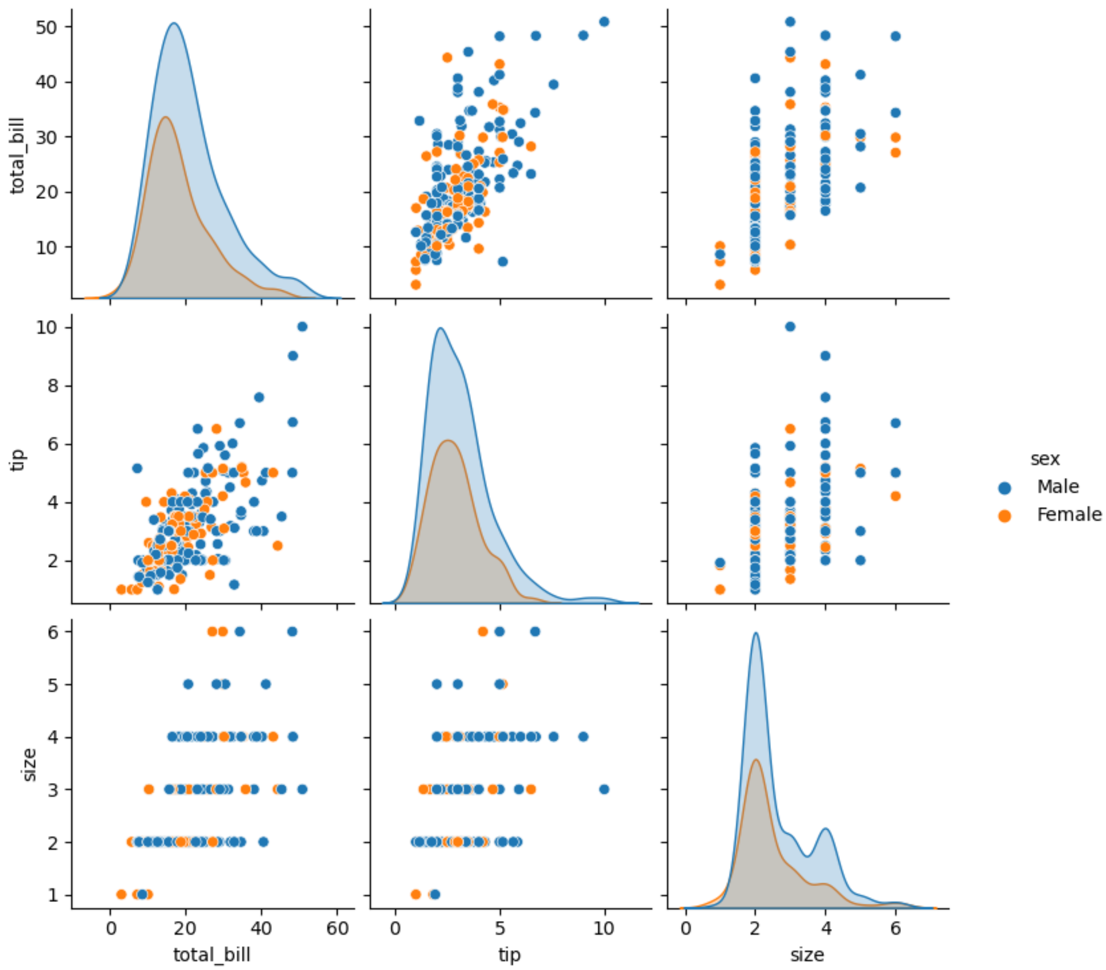
[29]:

	total_bill	tip	size
smoker			
Yes	20.756344	3.008710	2.408602
No	19.188278	2.991854	2.668874

(3.4) Pairplot Try to create a pairplot (check out the documentation in the link) where you differentiate between male and female, i.e. use `'sex'` to plot these aspecets in two different colours.

[30]:
```
sns.pairplot(data=tips, hue="sex")
plt.show()
```

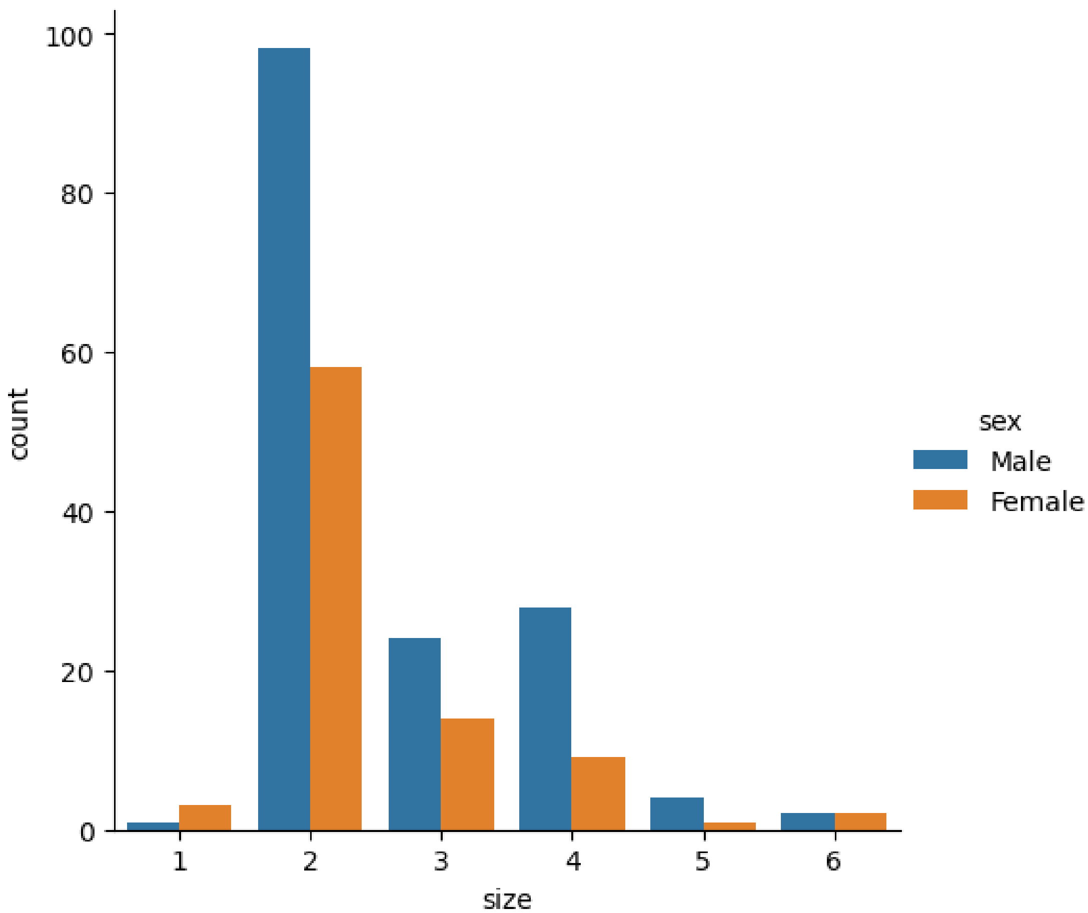
```
/Users/samarinm/anaconda3/envs/pythonCC/lib/python3.11/site-
packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to
tight
  self._figure.tight_layout(*args, **kwargs)
```



(3.5) Category Plots Try to create a catplot (check out the documentation in the link), which plots the `'size'` on the x-axis and the **count** of different group sizes on the y-axis (i.e. a histogram) and differentiate between male and female, i.e. use `'sex'` to plot these aspecets in two different colours.

[31]:
```
sns.catplot(data=tips, x="size", kind="count", hue="sex")
plt.show()
```

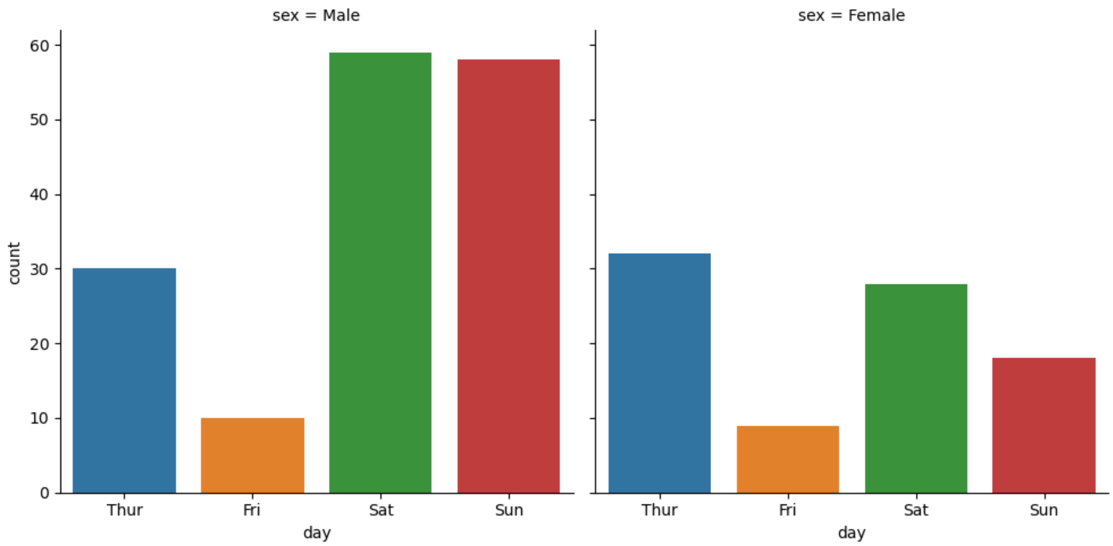
```
/Users/samarinm/anaconda3/envs/pythonCC/lib/python3.11/site-
packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to
tight
  self._figure.tight_layout(*args, **kwargs)
```



Now, try to create a catplot (check out the documentation in the link), which plots the `'day'` on the x-axis and the **count** of different group sizes on the y-axis (i.e. a histogram). This time create two subplots, one for female and one for male, i.e. use `'sex'` for the plot **columns**.

[32]:
```
sns.catplot(data=tips, x="day", kind="count", col="sex")
plt.show()
```

```
/Users/samarinm/anaconda3/envs/pythonCC/lib/python3.11/site-
packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to
tight
  self._figure.tight_layout(*args, **kwargs)
```



(3.6) Correlation In this last exercise, we compute the correlation between the numerical variables. To this end, use the method `corr()` and put your solution in the following cell:

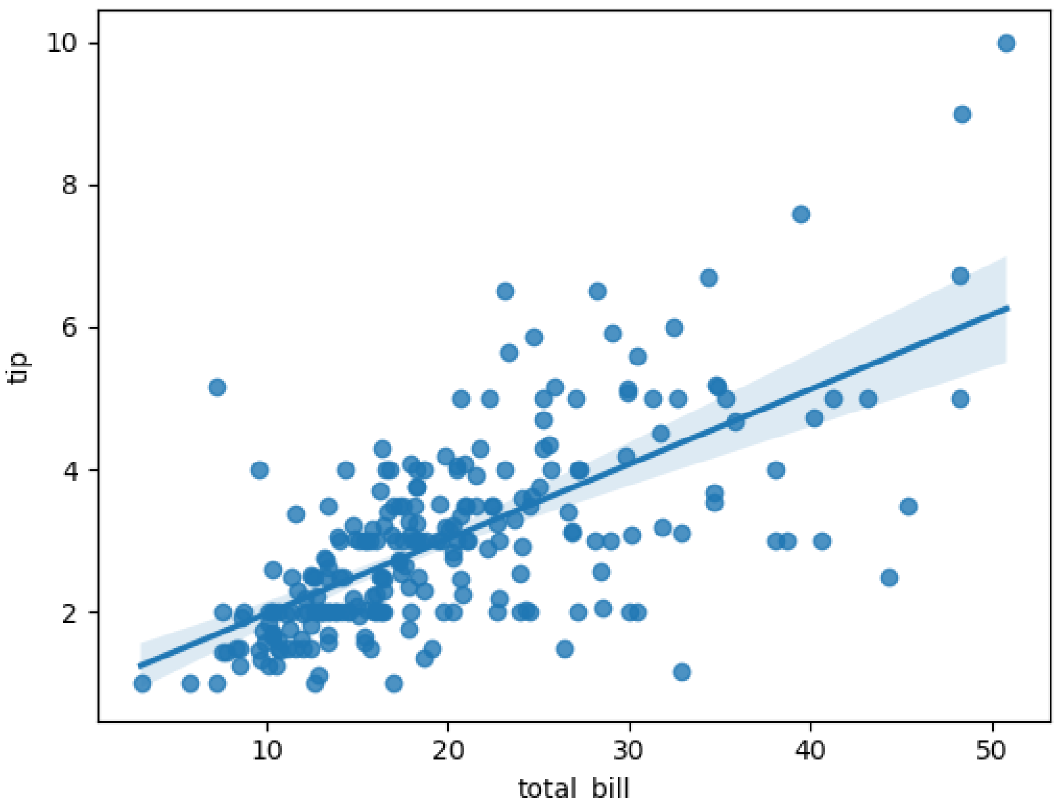
[35]:
```
tips.corr(numeric_only=True)
```

[35]:

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

Try to create a regplot (check out the documentation in the link), which plots the `'total_bill'` on the x-axis and the `'tip'` on the y-axis.

[34]:
```
sns.regplot(data=tips, x="total_bill", y="tip", ci=95)
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill