MODULE P02-INTRODUCTION-PYTHON

Welcome to the course "CPP Data Science & AI"!

DATA INFORMATIE KENNIS WIJSHEID

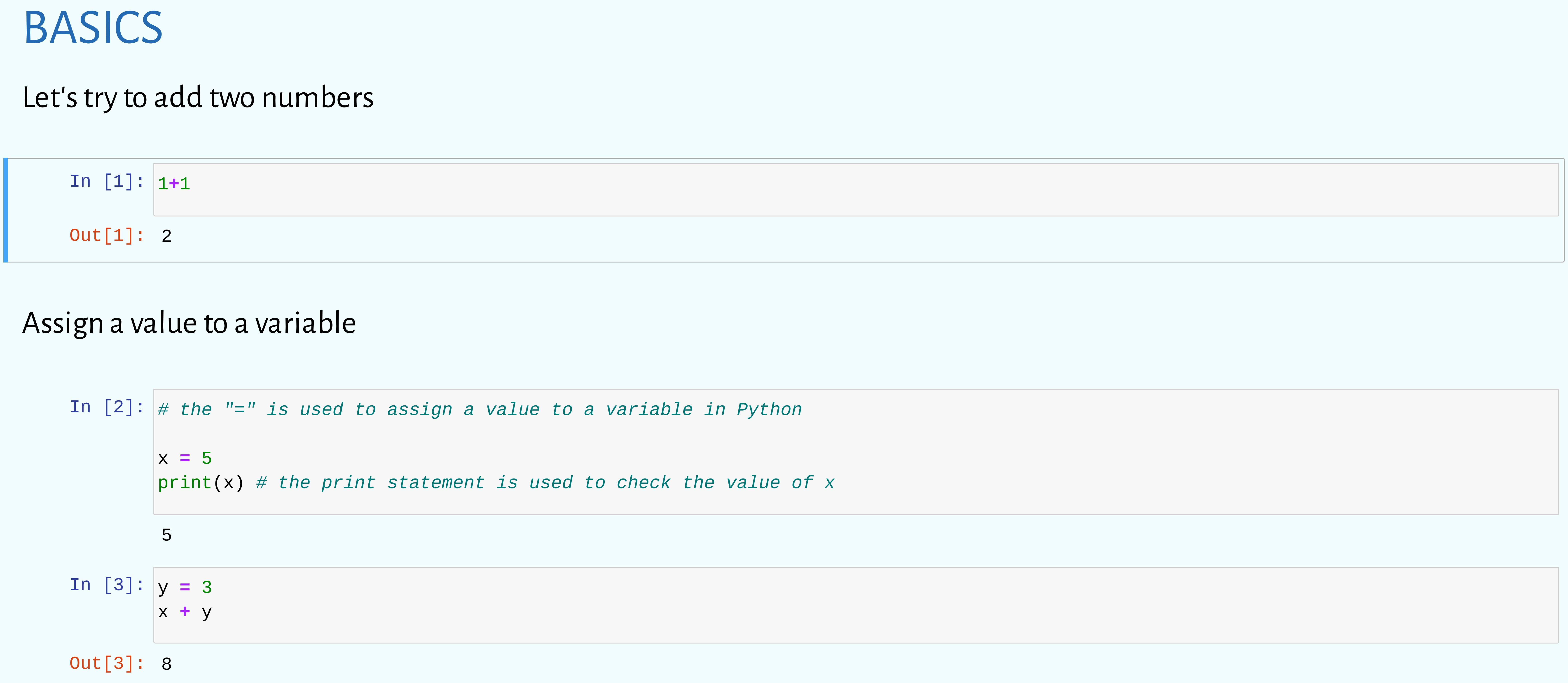
JUPYTER NOTEBOOKS

During this course, you will learn Python via Jupyter notebooks.

Jupyter notebooks are composed of cells. In each cell, you can write text, or code chunks. When running a cell, the code will be executed and its output displayed.

Jupyter notebooks are interactive, which means that you can easily modify the contents of a cell, and see whether the output is to your liking or not.

Another advantage of Jupyter notebooks is that they can be converted to a slidedeck. All presentations in this course are made by converting Jupyter notebooks to a slidedeck.



BASICS

Let's try to add two numbers

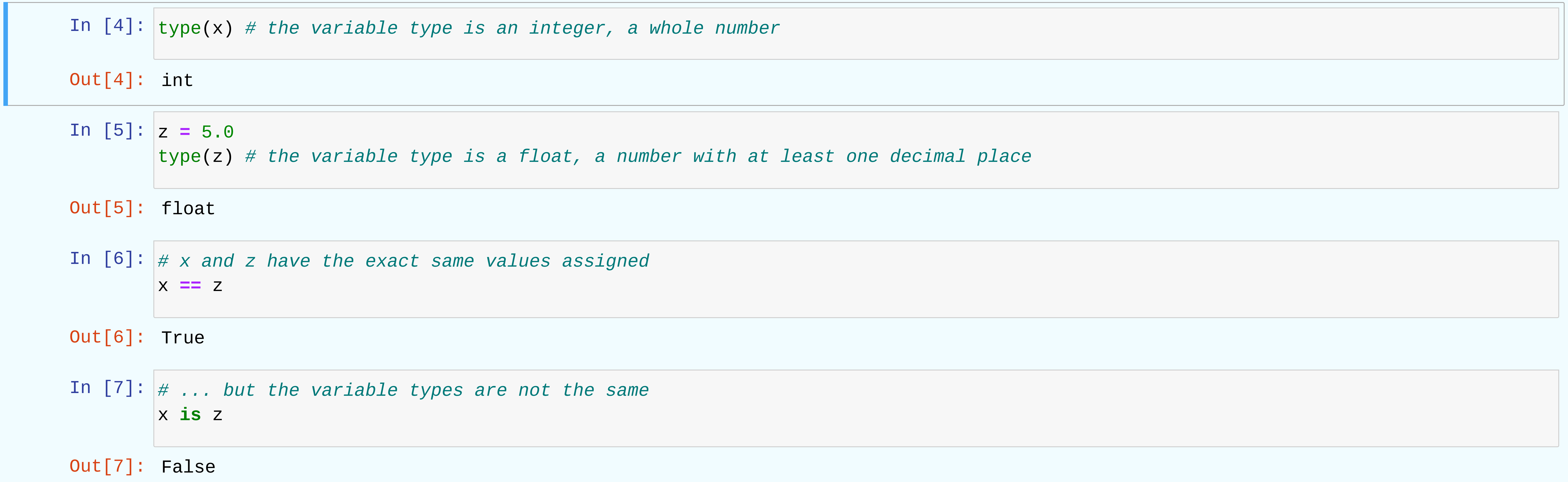
```
In [1]: 1+1

Out[1]: 2
```

Assign a value to a variable

```
In [2]: # the "=" is used to assign a value to a variable in Python

        x = 5
        print(x) # the print statement is used to check the value of x

        5

In [3]: y = 3
        x + y

Out[3]: 8
```

VARIABLE TYPES

With the type() function, you can check the variable type. When assigning a value to a variable, you do not have to declare the variable upfront. Python will set the variable type for you, based on the value assigned to the variable.



```
In [4]:  type(x) # the variable type is an integer, a whole number

Out[4]:  int

In [5]:  z = 5.0
         type(z) # the variable type is a float, a number with at least one decimal place

Out[5]:  float

In [6]:  # x and z have the exact same values assigned
         x == z

Out[6]:  True

In [7]:  # ... but the variable types are not the same
         x is z

Out[7]:  False
```

PYTHON OBJECT TYPES

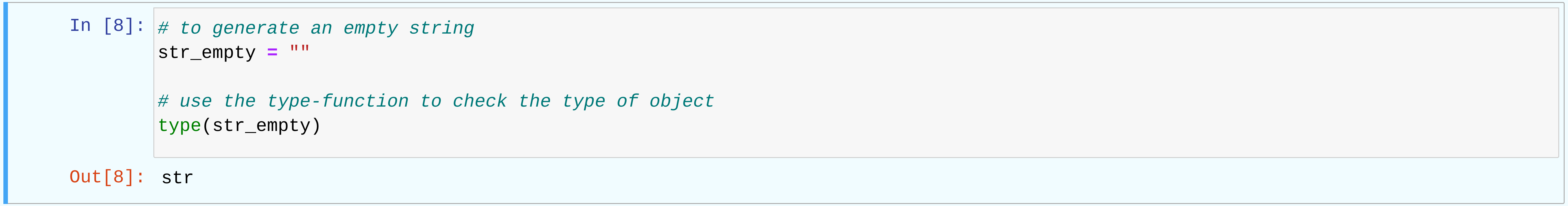
4 important Python objects are:

- strings
- lists
- dictionaries
- tuples

In this module, we will explain strings, lists and dictionaries.

PYTHON OBJECT TYPE 1: STRINGS

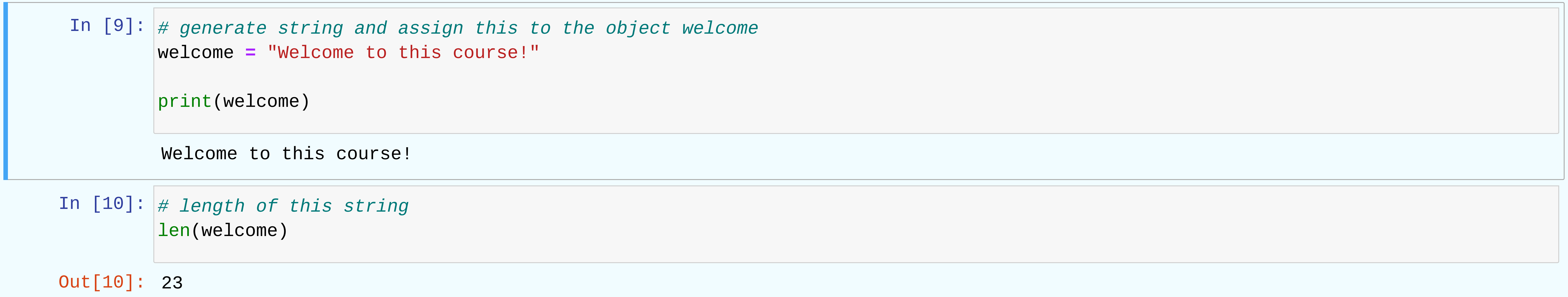
A string is a sequence of characters.



```
In [8]: # to generate an empty string
        str_empty = ""

        # use the type-function to check the type of object
        type(str_empty)

Out[8]: str
```

An example is the string "Welcome to this course!"

The function len() returns the length of a string. Please note that whitespaces and punctuation are also counted as characters.



In [9]:
```
# generate string and assign this to the object welcome
welcome = "Welcome to this course!"

print(welcome)
```

```
Welcome to this course!
```

In [10]:
```
# length of this string
len(welcome)
```

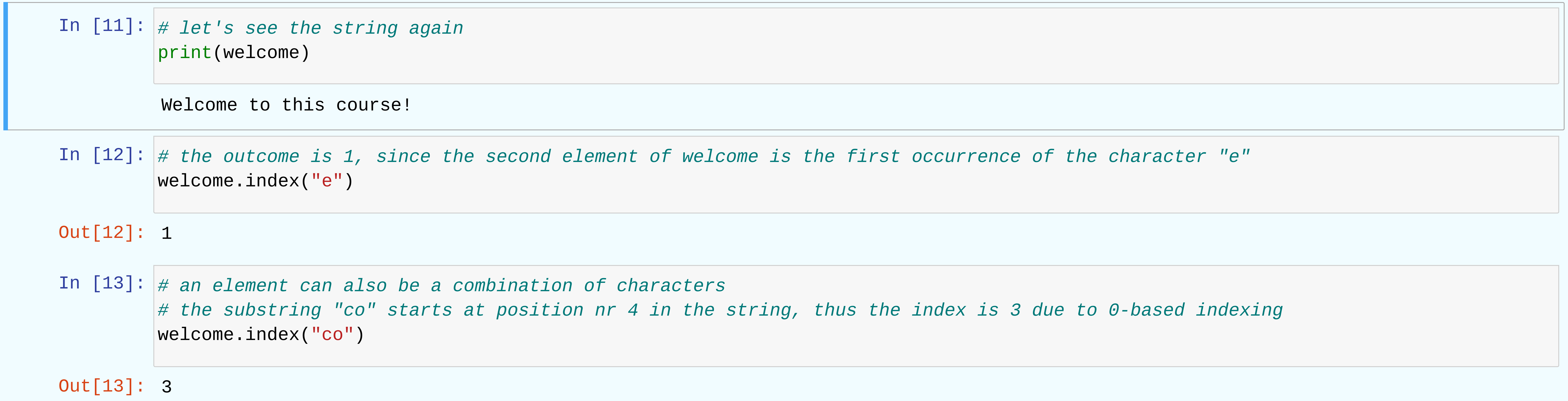
Out[10]:
```
23
```

INDEXING

An index refers to a position in an ordered list. A string can be seen as a list of characters.

The function index() used on a string returns the position of the first occurrence of the element in that string, i.e. the lowest index for this element.

Python uses 0-based indexing, which means that index = 0 refers to the first element, index = 1 to the second element, etc.



```
In [11]: # let's see the string again
         print(welcome)

         Welcome to this course!

In [12]: # the outcome is 1, since the second element of welcome is the first occurrence of the character "e"
         welcome.index("e")

Out[12]: 1

In [13]: # an element can also be a combination of characters
         # the substring "co" starts at position nr 4 in the string, thus the index is 3 due to 0-based indexing
         welcome.index("co")

Out[13]: 3
```

SLICING

Now we do the opposite: let's use indices to get a particular substring, a set of sequential characters, from a string

In general, slicing has the following form:

a[start:stop]

where:

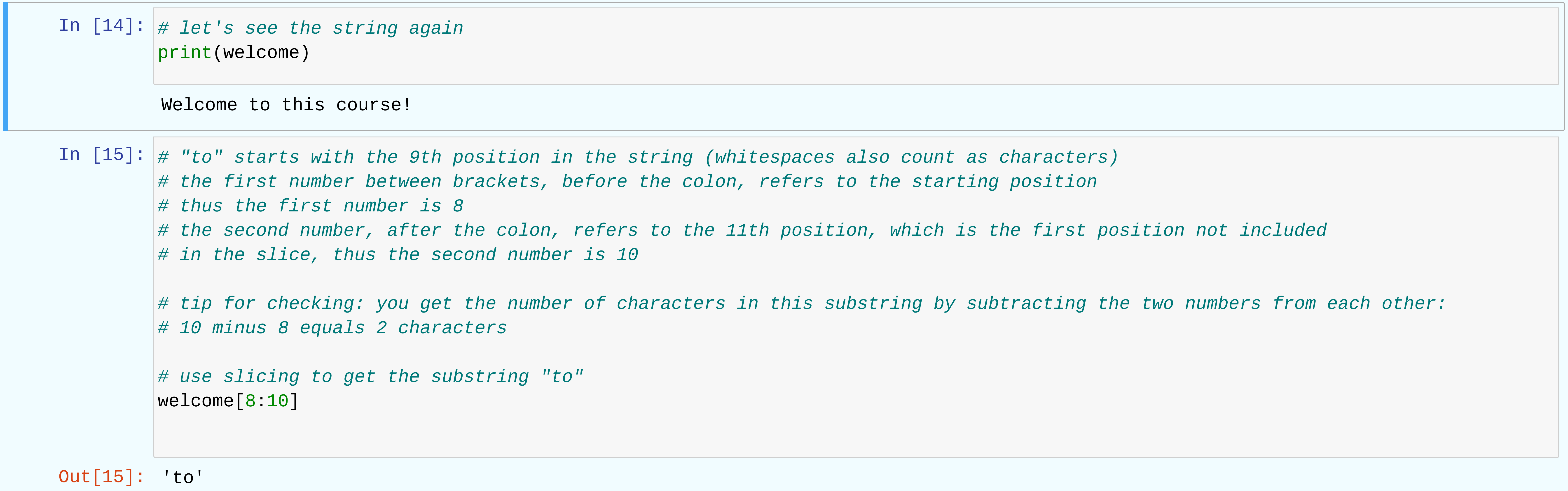
- a = an object (for now, we start with slicing a string)
- start = the starting position of the first character in the substring
- stop = the position of the first character which is NOT included in the selected slice

The round brackets are used for functions, squared brackets are used for slicing

From: https://stackoverflow.com/questions/509211/understanding-slice-notation

Let's use indices to get a particular word from our welcome string, in this example the word "to"

Remember, Python used 0-based indexing, e.g. index 8 refers to the 9th position in the string

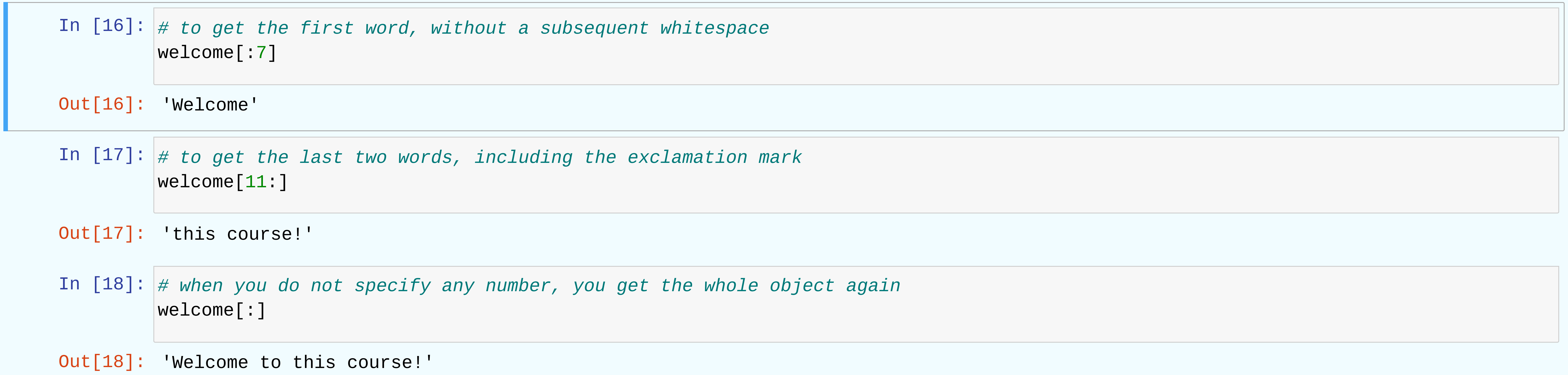


```
In [14]: # let's see the string again
         print(welcome)

         Welcome to this course!
```

```
In [15]: # "to" starts with the 9th position in the string (whitespaces also count as characters)
         # the first number between brackets, before the colon, refers to the starting position
         # thus the first number is 8
         # the second number, after the colon, refers to the 11th position, which is the first position not included
         # in the slice, thus the second number is 10

         # tip for checking: you get the number of characters in this substring by subtracting the two numbers from each other:
         # 10 minus 8 equals 2 characters

         # use slicing to get the substring "to"
         welcome[8:10]

Out[15]: 'to'
```

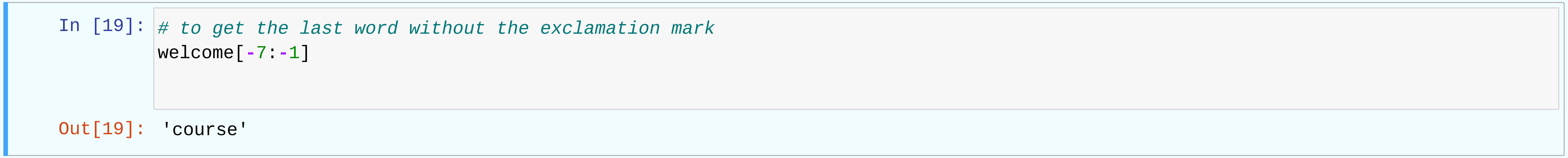
For slicing, you do not need to specify (both) numbers.

- a[start:] - the slice starts at the specified index, and includes the rest of the array
- a[:stop] - the slice starts at the beginning of the string, and stops at the (stop-1)th position



```
In [16]: # to get the first word, without a subsequent whitespace
         welcome[:7]

Out[16]: 'Welcome'

In [17]: # to get the last two words, including the exclamation mark
         welcome[11:]

Out[17]: 'this course!'

In [18]: # when you do not specify any number, you get the whole object again
         welcome[:]

Out[18]: 'Welcome to this course!'
```

Negative numbers can also be used for slicing



```
In [19]: # to get the last word without the exclamation mark
         welcome[-7:-1]

Out[19]: 'course'
```

EXPLICIT STEP-ARGUMENT IN SLICING

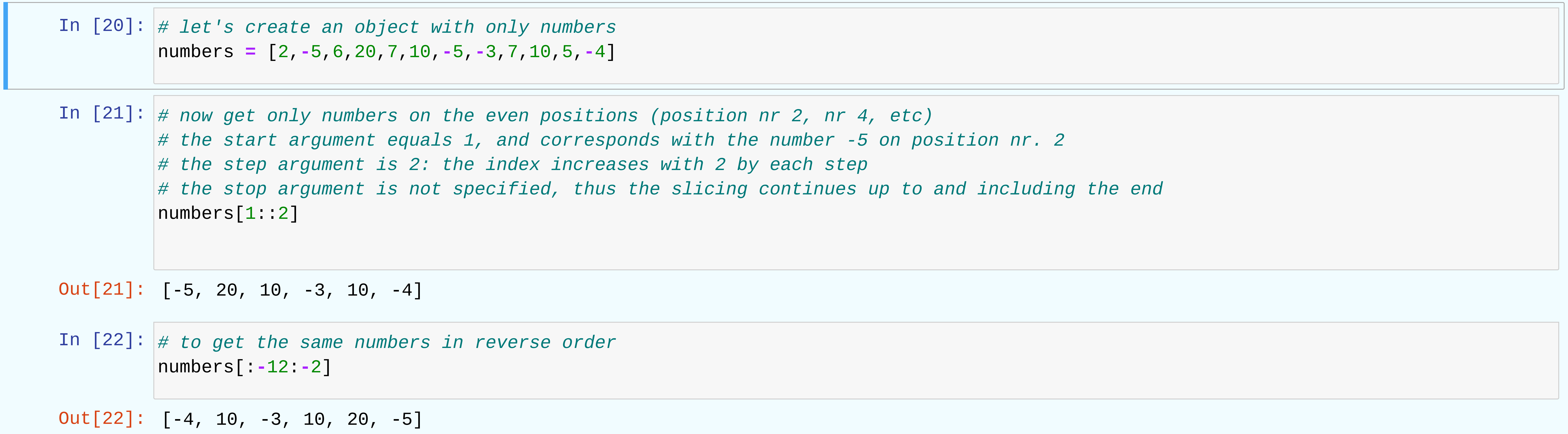
You can also specify the step-argument for slicing.

a[start:stop:step]

where:

- a = an object
- start = the starting position of the first element
- stop = the position of the first element which is NOT included in the selected slice
- step = the amount by which the index increases per step. When the step argument is not specified, the default is 1

From: https://stackoverflow.com/questions/509211/understanding-slice-notation



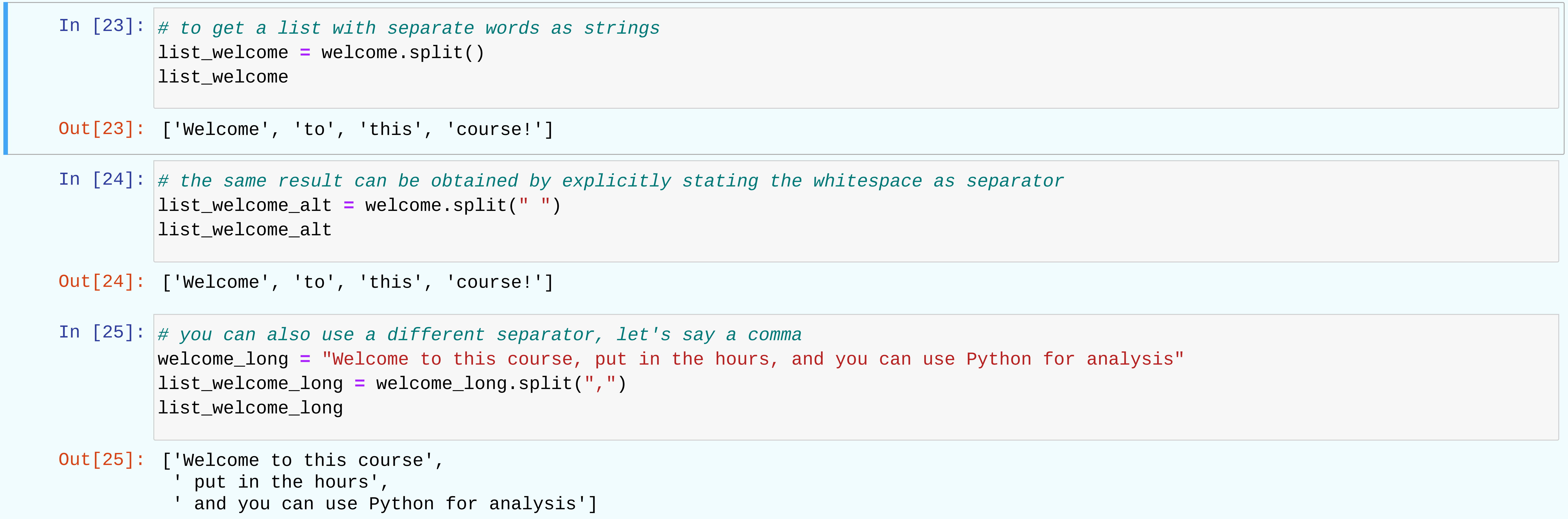
```
In [20]: # let's create an object with only numbers
         numbers = [2,-5,6,20,7,10,-5,-3,7,10,5,-4]
```

```
In [21]: # now get only numbers on the even positions (position nr 2, nr 4, etc)
         # the start argument equals 1, and corresponds with the number -5 on position nr. 2
         # the step argument is 2: the index increases with 2 by each step
         # the stop argument is not specified, thus the slicing continues up to and including the end
         numbers[1::2]

Out[21]: [-5, 20, 10, -3, 10, -4]
```

```
In [22]: # to get the same numbers in reverse order
         numbers[:-12:-2]

Out[22]: [-4, 10, -3, 10, 20, -5]
```

MOVING FROM STRINGS TO LISTS

Our infamous welcome string contains several words. The split-function returns a list of strings. By default, the whitespace is used as separator, and the resulting list contains strings, each containing one word.



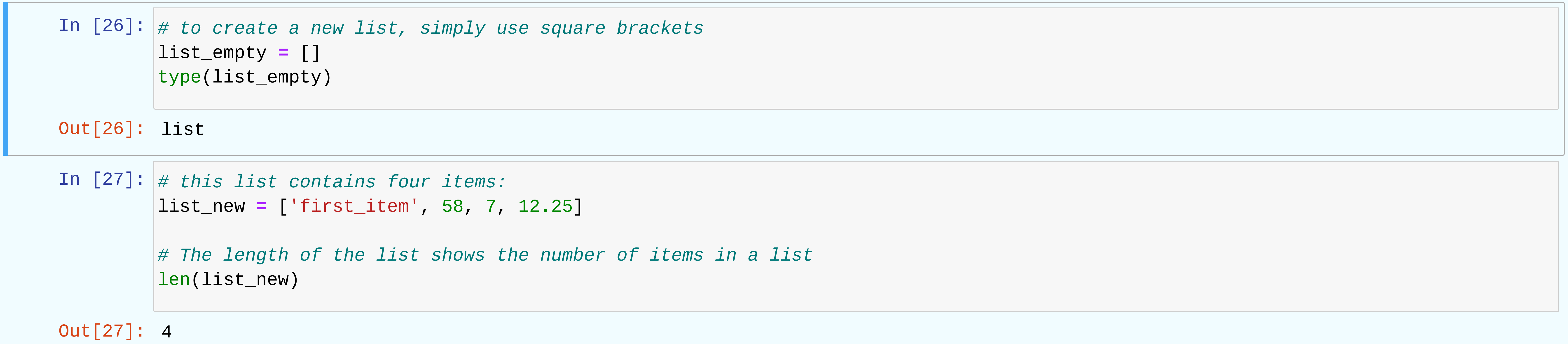
```
In [23]: # to get a list with separate words as strings
         list_welcome = welcome.split()
         list_welcome

Out[23]: ['Welcome', 'to', 'this', 'course!']

In [24]: # the same result can be obtained by explicitly stating the whitespace as separator
         list_welcome_alt = welcome.split(" ")
         list_welcome_alt

Out[24]: ['Welcome', 'to', 'this', 'course!']

In [25]: # you can also use a different separator, let's say a comma
         welcome_long = "Welcome to this course, put in the hours, and you can use Python for analysis"
         list_welcome_long = welcome_long.split(",")
         list_welcome_long

Out[25]: ['Welcome to this course',
          ' put in the hours',
          ' and you can use Python for analysis']
```

This results in a list containing three elements, substrings from the original string.

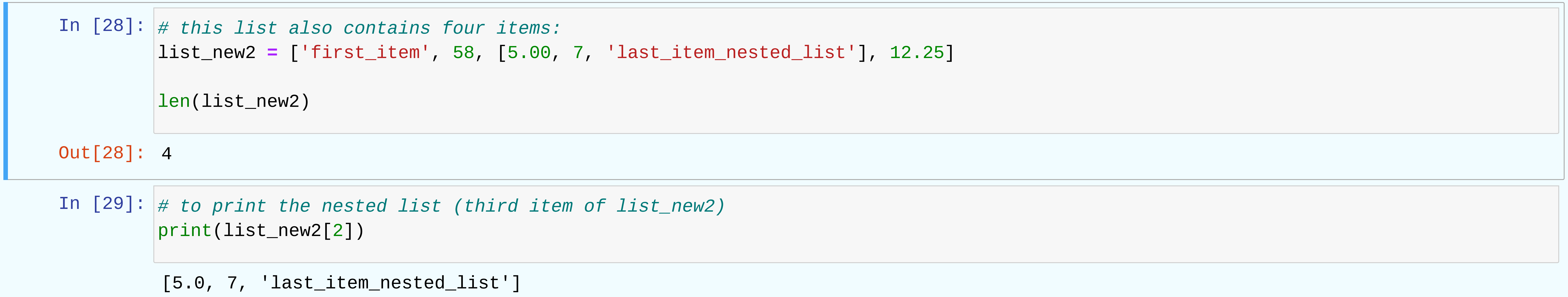
PYTHON OBJECT TYPE 2: LISTS

In Python, a list is an ordered sequence of items. The items of a list are put between square brackets, a comma is used to separate items from each other.

Lists are very flexible, they can contain items of various data types, and lists can also contain other lists (the lists within lists are nested lists)



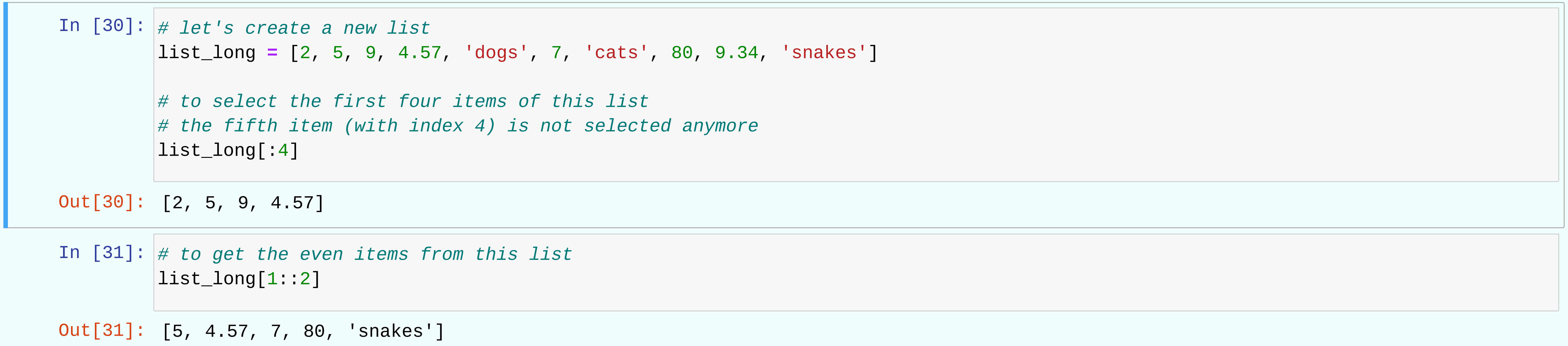
```
In [26]: # to create a new list, simply use square brackets
         list_empty = []
         type(list_empty)

Out[26]: list

In [27]: # this list contains four items:
         list_new = ['first_item', 58, 7, 12.25]

         # The length of the list shows the number of items in a list
         len(list_new)

Out[27]: 4
```



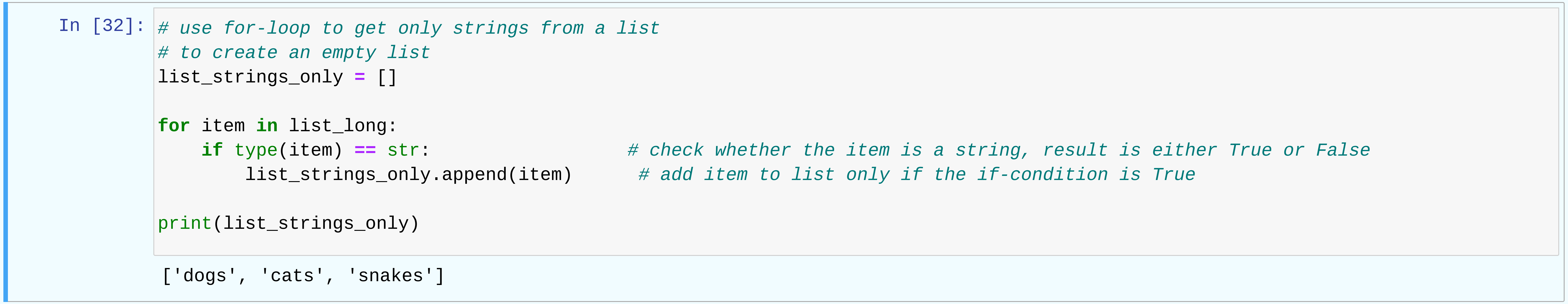
```
In [28]: # this list also contains four items:
         list_new2 = ['first_item', 58, [5.00, 7, 'last_item_nested_list'], 12.25]

         len(list_new2)

Out[28]: 4

In [29]: # to print the nested list (third item of list_new2)
         print(list_new2[2])

         [5.0, 7, 'last_item_nested_list']
```

SLICING LIST

Lists can be sliced in similar ways as strings.



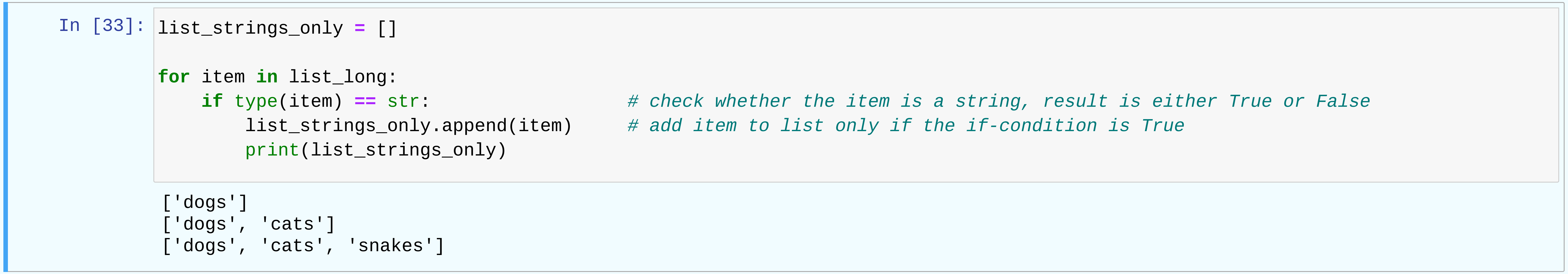
```
In [30]: # let's create a new list
         list_long = [2, 5, 9, 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes']

         # to select the first four items of this list
         # the fifth item (with index 4) is not selected anymore
         list_long[:4]

Out[30]: [2, 5, 9, 4.57]

In [31]: # to get the even items from this list
         list_long[1::2]

Out[31]: [5, 4.57, 7, 80, 'snakes']
```



```
In [32]: # use for-loop to get only strings from a list
         # to create an empty list
         list_strings_only = []

         for item in list_long:
             if type(item) == str:                  # check whether the item is a string, result is either True or False
                 list_strings_only.append(item)      # add item to list only if the if-condition is True

         print(list_strings_only)

          ['dogs', 'cats', 'snakes']
```

INDENTATION IN LOOPS

In Python, indentation is important. Compare the output from this code block to the previous slide. At the end of every iteration, the list is printed. Since we use append(), we can see that an item is added to the list when the condition is true. This is because the print-statement is indented within the if-function.

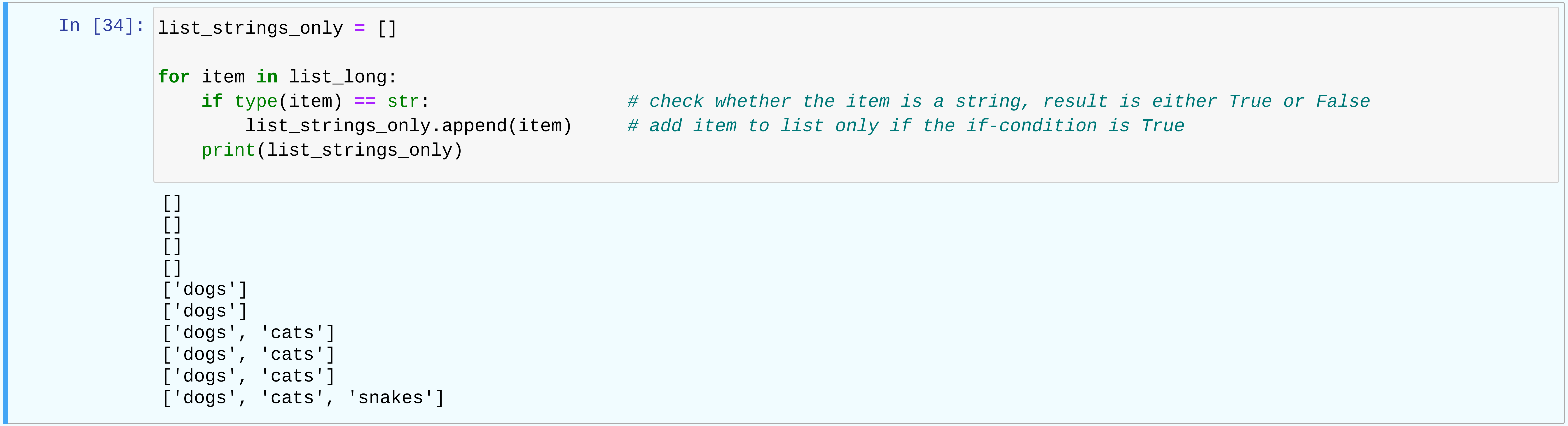


In [33]:

```
list_strings_only = []

for item in list_long:
    if type(item) == str:                  # check whether the item is a string, result is either True or False
        list_strings_only.append(item)     # add item to list only if the if-condition is True
        print(list_strings_only)
```

```
['dogs']
['dogs', 'cats']
['dogs', 'cats', 'snakes']
```

Can you explain the following result?



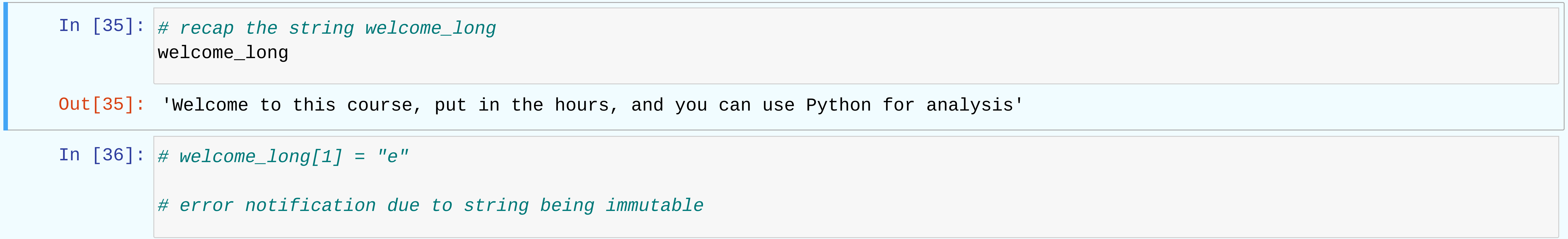
In [34]:

```
list_strings_only = []

for item in list_long:
    if type(item) == str:                    # check whether the item is a string, result is either True or False
        list_strings_only.append(item)       # add item to list only if the if-condition is True
    print(list_strings_only)
```

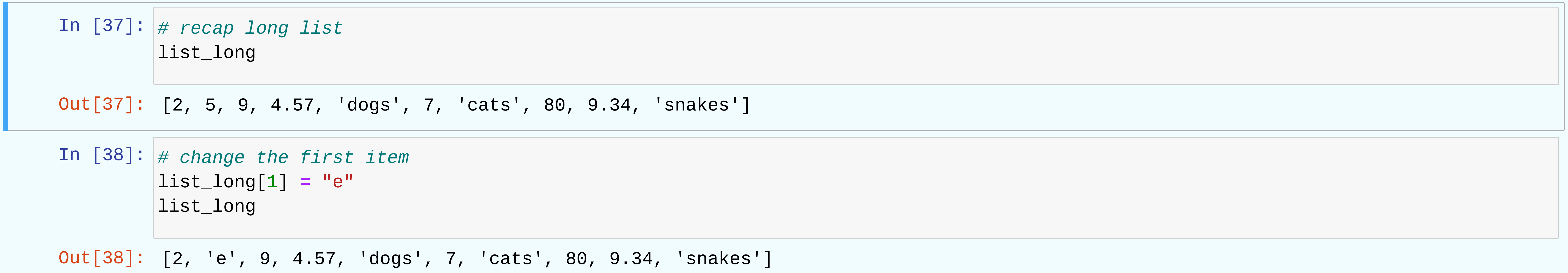
```
[]
[]
[]
[]
['dogs']
['dogs']
['dogs', 'cats']
['dogs', 'cats']
['dogs', 'cats']
['dogs', 'cats', 'snakes']
```

DIFFERENCES BETWEEN LISTS AND STRINGS

Lists are mutable, strings are not.

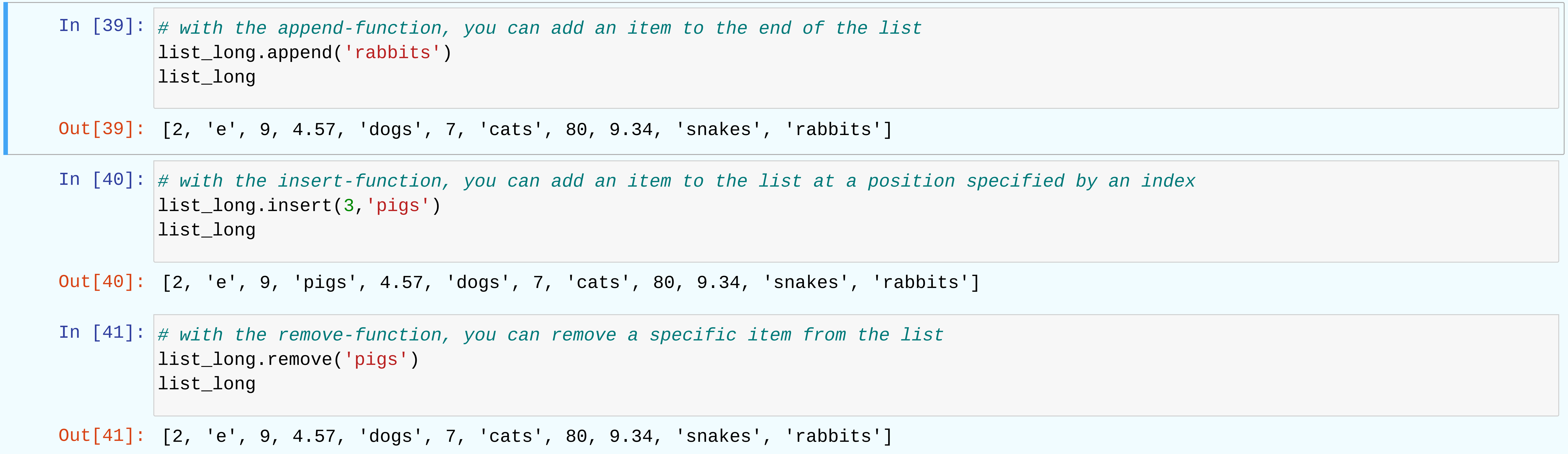


```
In [35]: # recap the string welcome_long
         welcome_long

Out[35]: 'Welcome to this course, put in the hours, and you can use Python for analysis'

In [36]: # welcome_long[1] = "e"

         # error notification due to string being immutable
```



```
In [37]: # recap long list
         list_long

Out[37]: [2, 5, 9, 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes']

In [38]: # change the first item
         list_long[1] = "e"
         list_long

Out[38]: [2, 'e', 9, 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes']
```

FUNCTIONS FOR MUTABLE LISTS

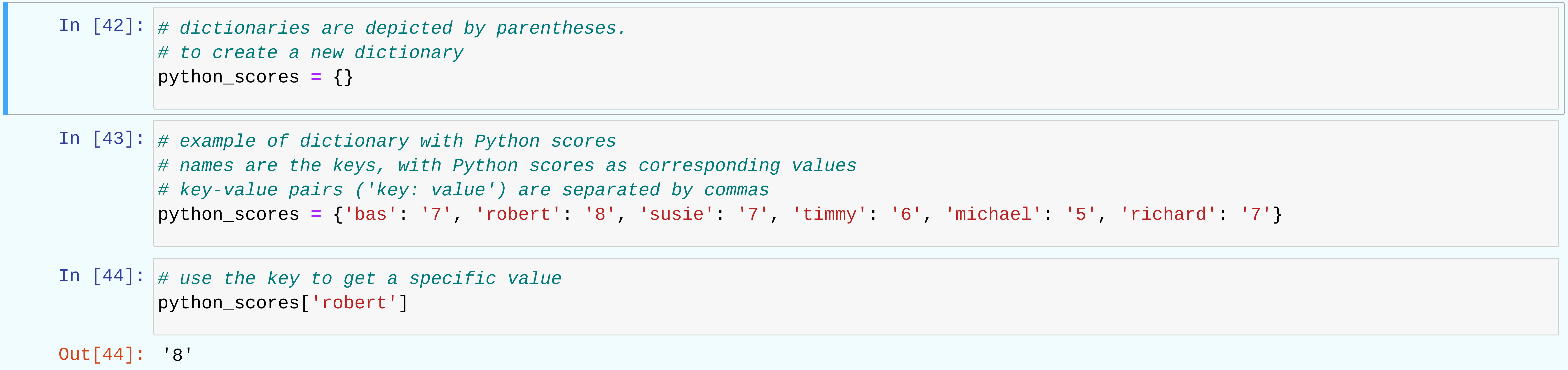


```
In [39]: # with the append-function, you can add an item to the end of the list
         list_long.append('rabbits')
         list_long

Out[39]: [2, 'e', 9, 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes', 'rabbits']

In [40]: # with the insert-function, you can add an item to the list at a position specified by an index
         list_long.insert(3,'pigs')
         list_long

Out[40]: [2, 'e', 9, 'pigs', 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes', 'rabbits']

In [41]: # with the remove-function, you can remove a specific item from the list
         list_long.remove('pigs')
         list_long

Out[41]: [2, 'e', 9, 4.57, 'dogs', 7, 'cats', 80, 9.34, 'snakes', 'rabbits']
```

PYTHON OBJECT TYPE 3: DICTIONARIES

Dictionaries is a collection which is unordered and changeable.

Dictionaries contain key-value pairs, specific values can be looked up by using a key.

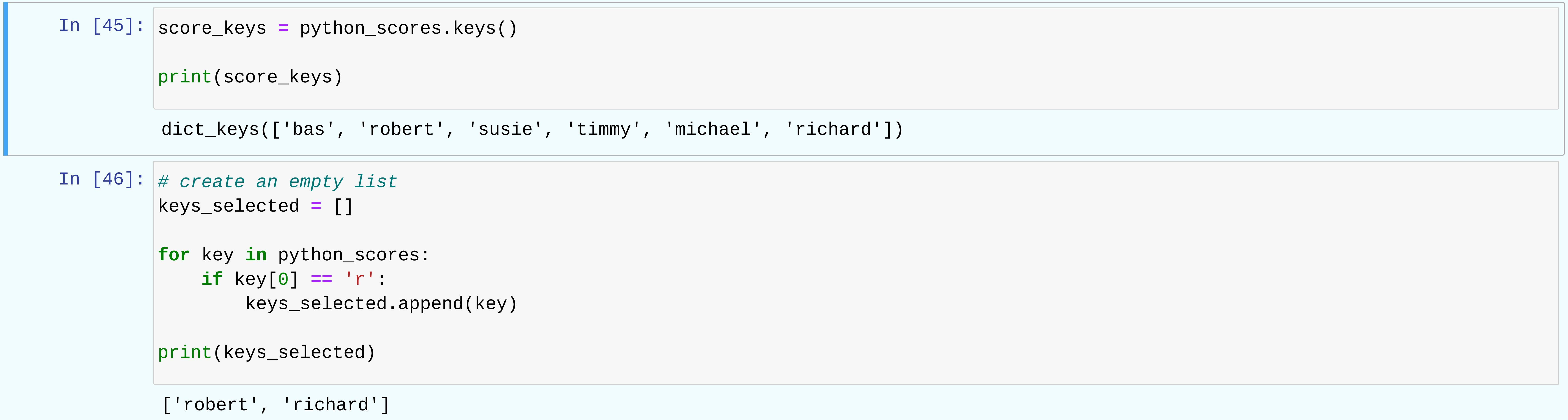


```
In [42]: # dictionaries are depicted by parentheses.
         # to create a new dictionary
         python_scores = {}

In [43]: # example of dictionary with Python scores
         # names are the keys, with Python scores as corresponding values
         # key-value pairs ('key: value') are separated by commas
         python_scores = {'bas': '7', 'robert': '8', 'susie': '7', 'timmy': '6', 'michael': '5', 'richard': '7'}

In [44]: # use the key to get a specific value
         python_scores['robert']

Out[44]: '8'
```

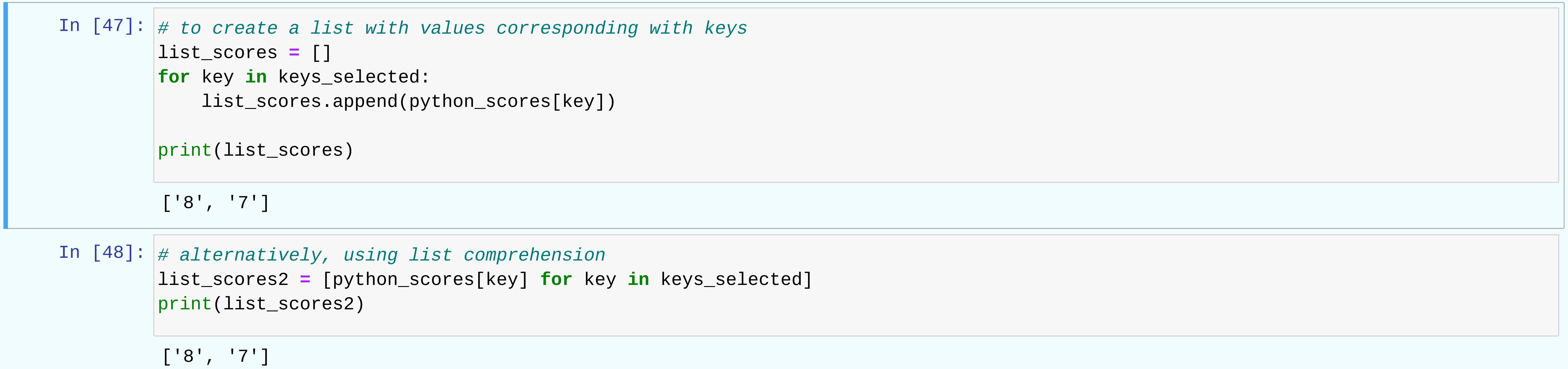
SLICING WITH DICTIONARIES…

Does not work: since a dictionary is not a sequence, we cannot slice a dictionary.

However, we can use a selection of keys to retrieve corresponding values.



```
In [45]: score_keys = python_scores.keys()

         print(score_keys)

         dict_keys(['bas', 'robert', 'susie', 'timmy', 'michael', 'richard'])

In [46]: # create an empty list
         keys_selected = []

         for key in python_scores:
             if key[0] == 'r':
                 keys_selected.append(key)

         print(keys_selected)

         ['robert', 'richard']
```



In [47]:

```
# to create a list with values corresponding with keys
list_scores = []
for key in keys_selected:
    list_scores.append(python_scores[key])

print(list_scores)
```

```
['8', '7']
```

In [48]:

```
# alternatively, using list comprehension
list_scores2 = [python_scores[key] for key in keys_selected]
print(list_scores2)
```

```
['8', '7']
```

INTERMEZZO: LIBRARIES AND FUNCTIONS

Python has standard built-in functions. However, quite often you need a specific function from a specific library.

Libraries need to be installed on your machine before you can use them.

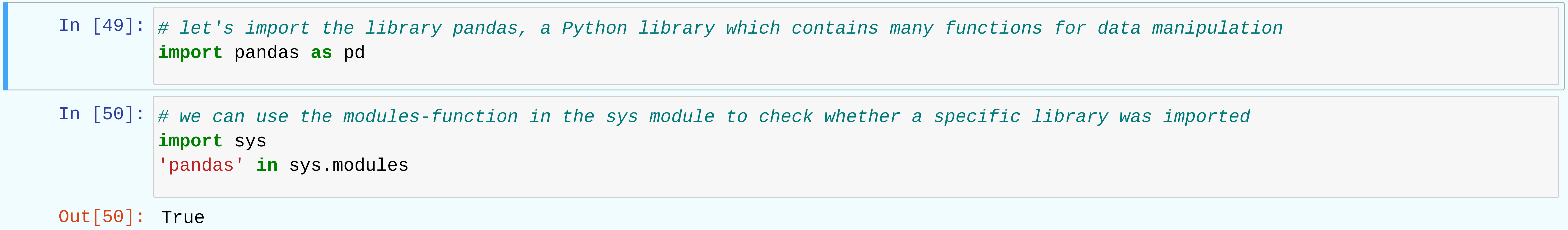
The standard way to install them is to use the Preferred Installer Program (pip)

To install a library, run the following code from the command line:

python -m pip install SomeLibrary

More information: https://docs.python.org/3/installing/index.html

After installment, you need to use the import statement to load a library



```
In [49]: # let's import the library pandas, a Python library which contains many functions for data manipulation
         import pandas as pd

In [50]: # we can use the modules-function in the sys module to check whether a specific library was imported
         import sys
         'pandas' in sys.modules

Out[50]: True
```

DATAFRAMES

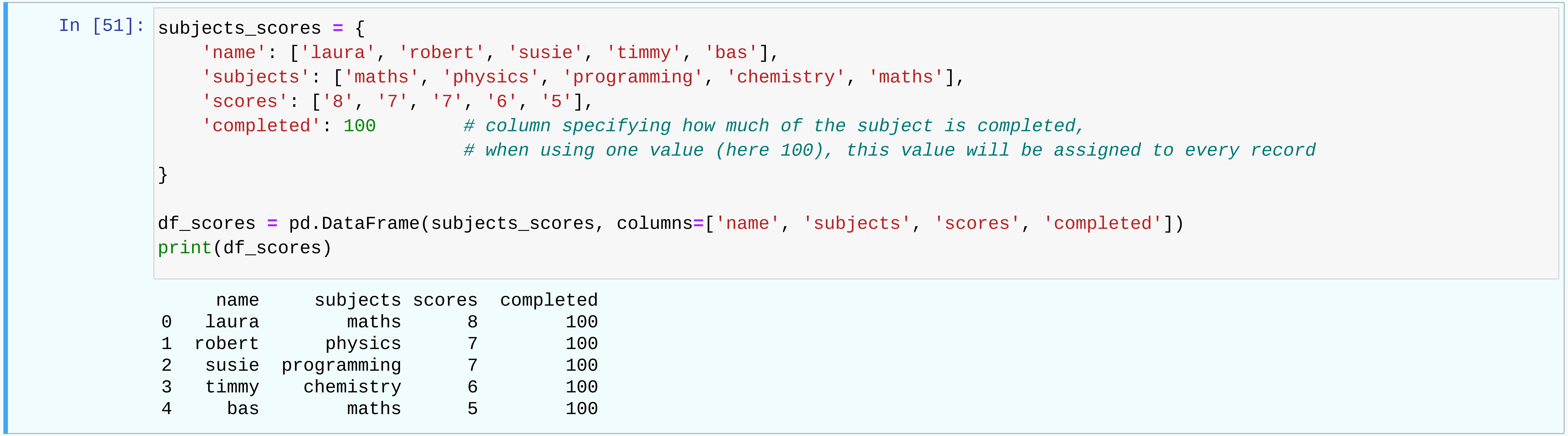
A dataframe is a 2-dimensional labeled data structure with columns of potentially different types. Data is aligned in a tabular fashion, with rows and columns. Rows have indices assigned to them, and columns are depicted by labels. A dataframe is a specific list, with columns of equal length.

Dataframe can be created in various ways with pandas:

- way 1: creating a DataFrame from various dictionaries
- way 2: creating a DataFrame from a list of dictionaries
- way 3: creating a DataFrame from reading files

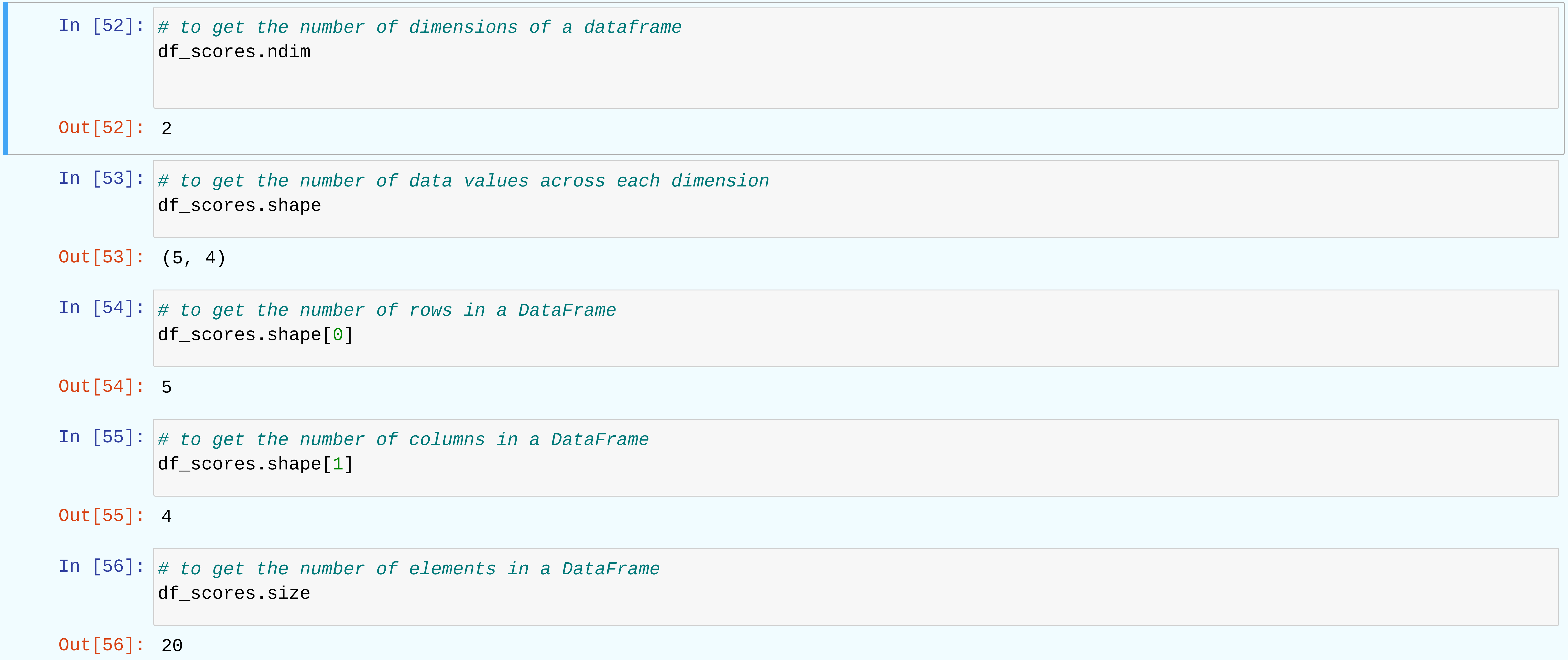
WAY 1: CREATING A DATAFRAME FROM VARIOUS DICTIONARIES

This way, the keys are the column labels, the dictionary values are the data values in the DataFrame



```
In [51]: subjects_scores = {
             'name': ['laura', 'robert', 'susie', 'timmy', 'bas'],
             'subjects': ['maths', 'physics', 'programming', 'chemistry', 'maths'],
             'scores': ['8', '7', '7', '6', '5'],
             'completed': 100        # column specifying how much of the subject is completed,
                                     # when using one value (here 100), this value will be assigned to every record
         }

         df_scores = pd.DataFrame(subjects_scores, columns=['name', 'subjects', 'scores', 'completed'])
         print(df_scores)
```

```
     name     subjects scores  completed
0   laura        maths      8        100
1  robert      physics      7        100
2   susie  programming      7        100
3   timmy    chemistry      6        100
4     bas        maths      5        100
```

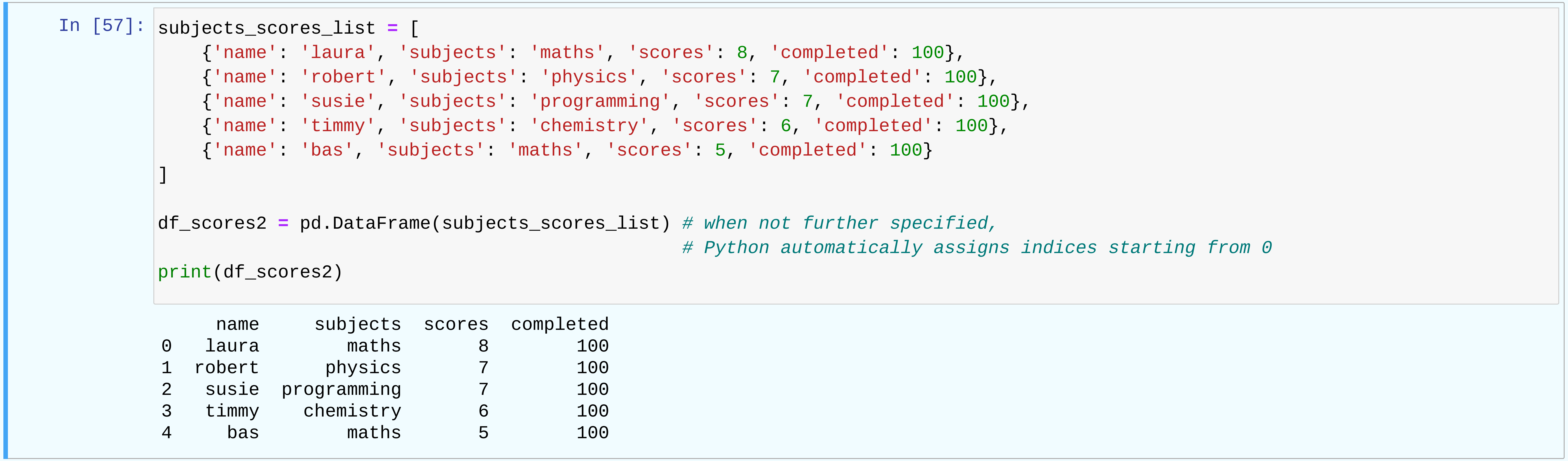


```
In [52]: # to get the number of dimensions of a dataframe
         df_scores.ndim

Out[52]: 2

In [53]: # to get the number of data values across each dimension
         df_scores.shape

Out[53]: (5, 4)

In [54]: # to get the number of rows in a DataFrame
         df_scores.shape[0]

Out[54]: 5

In [55]: # to get the number of columns in a DataFrame
         df_scores.shape[1]

Out[55]: 4

In [56]: # to get the number of elements in a DataFrame
         df_scores.size

Out[56]: 20
```

WAY 2: CREATING A DATAFRAME FROM A LIST OF DICTIONARIES

When having dictionaries with the same keys, you can create a dataframe by using the DataFrame function from the Pandas module.

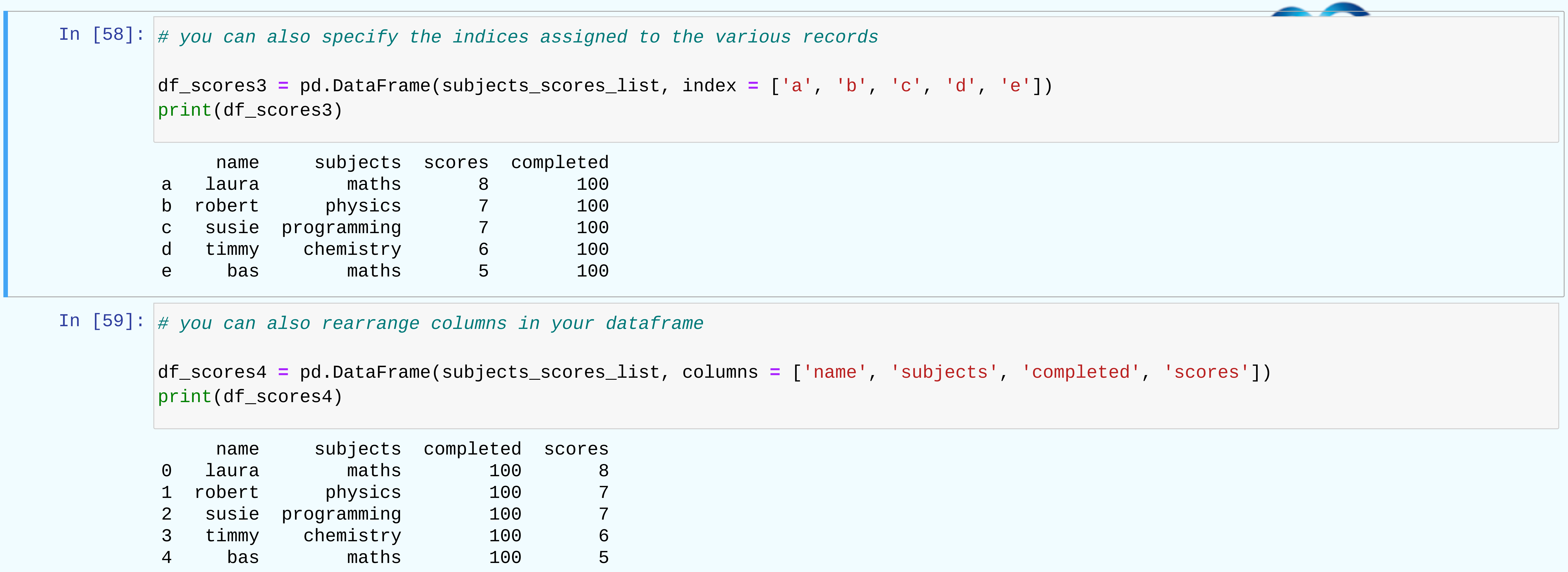
More info for next time: https://thispointer.com/pandas-create-dataframe-from-list-of-dictionaries/



In [57]:

```
subjects_scores_list = [
    {'name': 'laura', 'subjects': 'maths', 'scores': 8, 'completed': 100},
    {'name': 'robert', 'subjects': 'physics', 'scores': 7, 'completed': 100},
    {'name': 'susie', 'subjects': 'programming', 'scores': 7, 'completed': 100},
    {'name': 'timmy', 'subjects': 'chemistry', 'scores': 6, 'completed': 100},
    {'name': 'bas', 'subjects': 'maths', 'scores': 5, 'completed': 100}
]

df_scores2 = pd.DataFrame(subjects_scores_list) # when not further specified,
                                                # Python automatically assigns indices starting from 0
print(df_scores2)
```

```
     name     subjects  scores  completed
0   laura        maths       8        100
1  robert      physics       7        100
2   susie  programming       7        100
3   timmy    chemistry       6        100
4     bas        maths       5        100
```



```
In [58]: # you can also specify the indices assigned to the various records

         df_scores3 = pd.DataFrame(subjects_scores_list, index = ['a', 'b', 'c', 'd', 'e'])
         print(df_scores3)
```

```
     name     subjects  scores  completed
a   laura        maths       8        100
b  robert      physics       7        100
c   susie  programming       7        100
d   timmy    chemistry       6        100
e     bas        maths       5        100
```

```
In [59]: # you can also rearrange columns in your dataframe

         df_scores4 = pd.DataFrame(subjects_scores_list, columns = ['name', 'subjects', 'completed', 'scores'])
         print(df_scores4)
```

```
     name     subjects  completed  scores
0   laura        maths        100       8
1  robert      physics        100       7
2   susie  programming        100       7
3   timmy    chemistry        100       6
4     bas        maths        100       5
```

WAY 3: CREATING A DATAFRAME FROM READING FILES

A Comma Separated Values (CSV) file is an often used format. With the read_csv() function of the pandas module, you can directly read a dataset into dataframe format

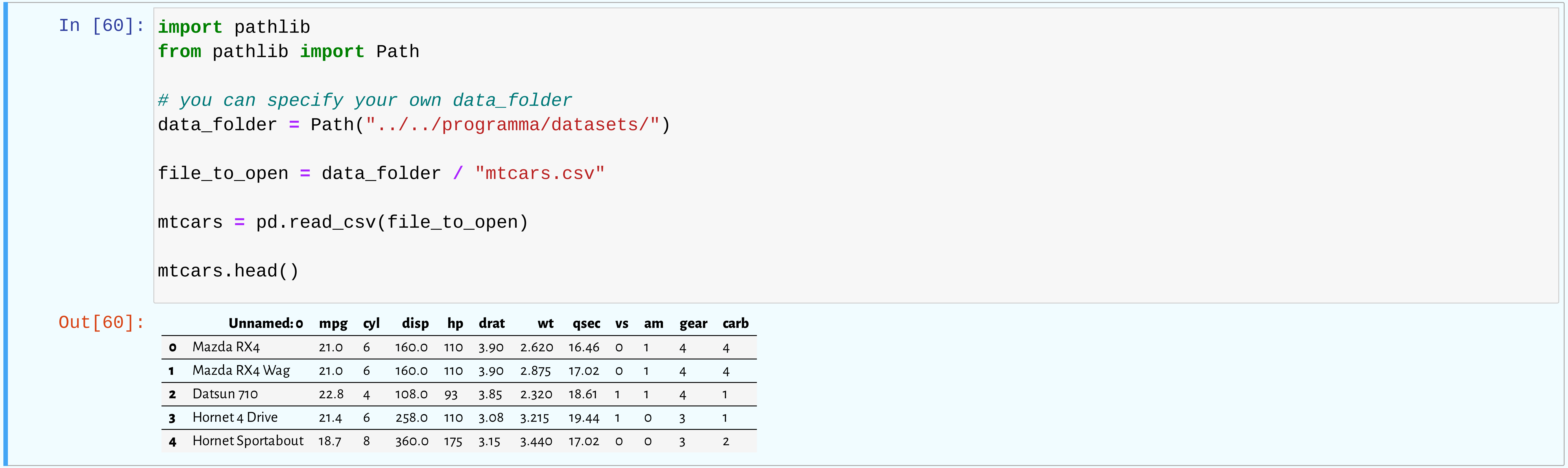
Further reading: https://realpython.com/pandas-read-write-files/#read-a-csv-file

EXAMPLE MTCARS

In the following slides, we will use the mtcars dataset to demonstrate functions.

Use the Pathlib module, in order for Python code to work on both Windows and Mac/Linux.

(https://medium.com/@ageitgey/python-3-quick-tip-the-easy-way-to-deal-with-file-paths-on-windows-mac-and-linux-11a072b58d5f)

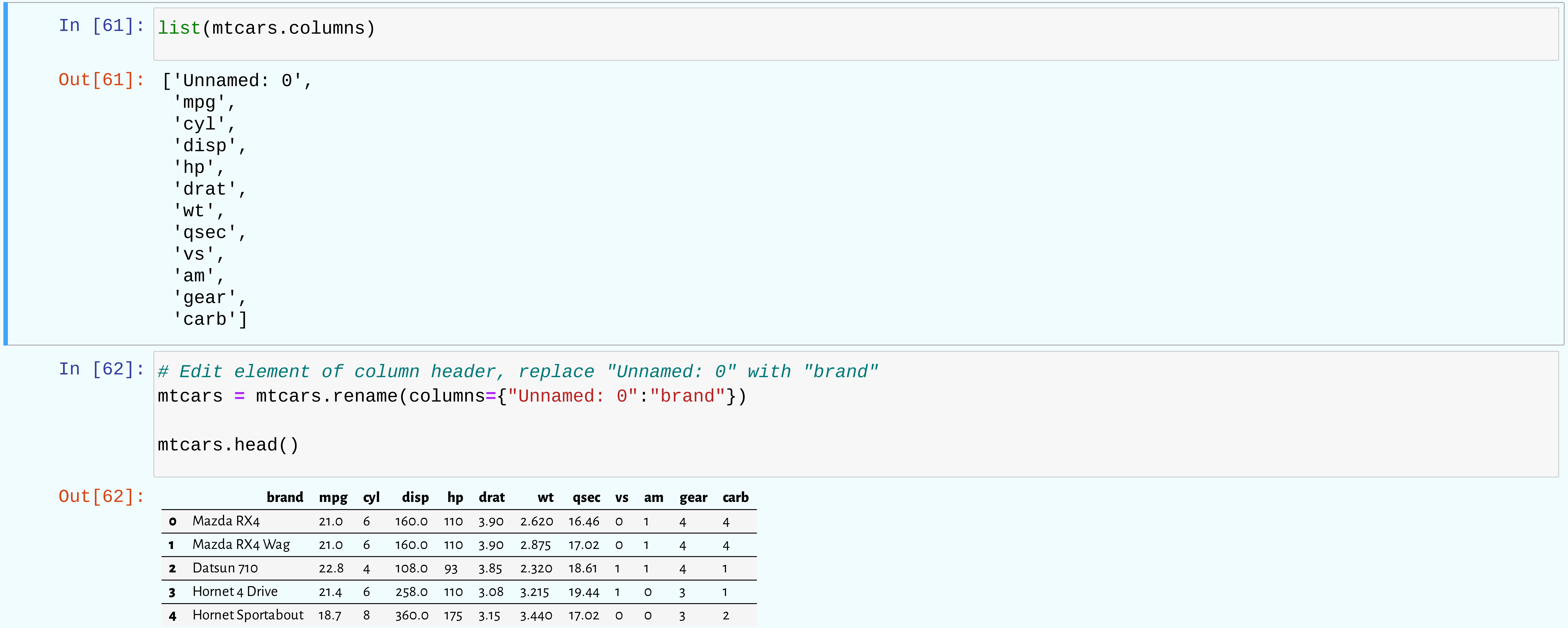


In [60]:

```python
import pathlib
from pathlib import Path

# you can specify your own data_folder
data_folder = Path("../../programma/datasets/")

file_to_open = data_folder / "mtcars.csv"

mtcars = pd.read_csv(file_to_open)

mtcars.head()
```

Out[60]:

	Unnamed: 0	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

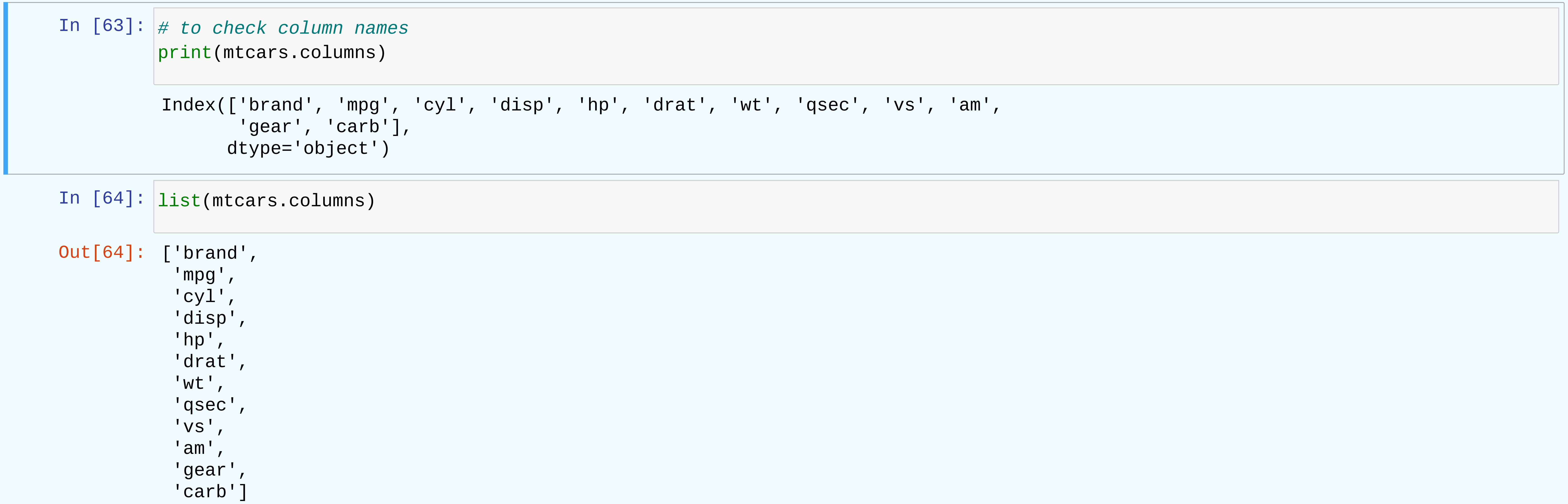


```
In [61]: list(mtcars.columns)

Out[61]: ['Unnamed: 0',
          'mpg',
          'cyl',
          'disp',
          'hp',
          'drat',
          'wt',
          'qsec',
          'vs',
          'am',
          'gear',
          'carb']
```

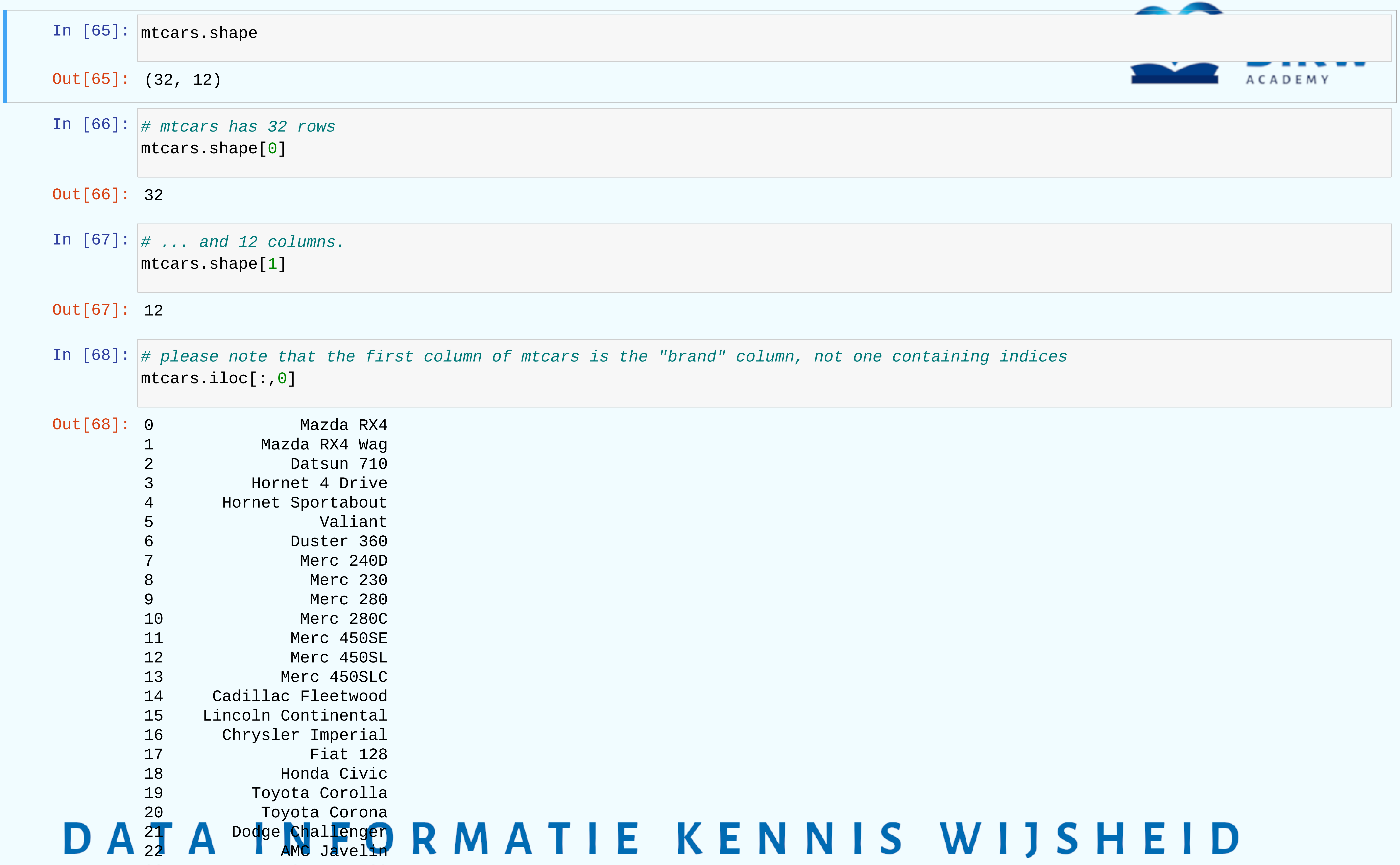
```
In [62]: # Edit element of column header, replace "Unnamed: 0" with "brand"
         mtcars = mtcars.rename(columns={"Unnamed: 0":"brand"})

         mtcars.head()
```

Out[62]:

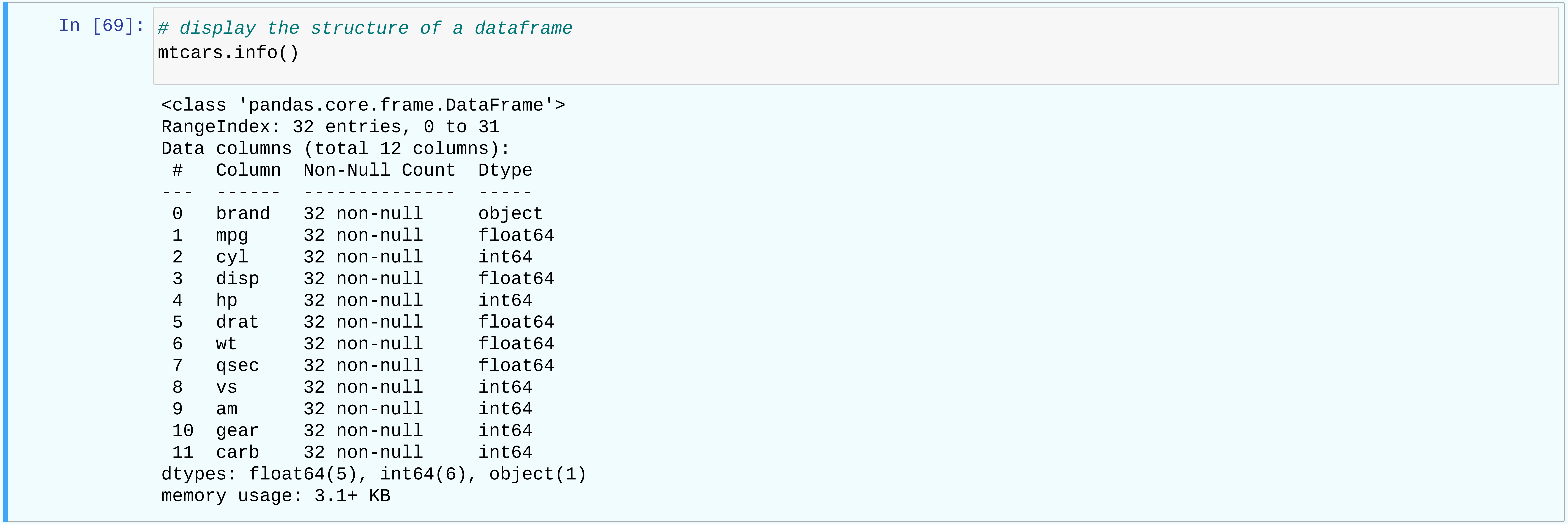
	brand	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2



```
In [63]: # to check column names
         print(mtcars.columns)

         Index(['brand', 'mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs', 'am',
                'gear', 'carb'],
               dtype='object')

In [64]: list(mtcars.columns)

Out[64]: ['brand',
          'mpg',
          'cyl',
          'disp',
          'hp',
          'drat',
          'wt',
          'qsec',
          'vs',
          'am',
          'gear',
          'carb']
```



```
In [65]: mtcars.shape

Out[65]: (32, 12)

In [66]: # mtcars has 32 rows
         mtcars.shape[0]

Out[66]: 32

In [67]: # ... and 12 columns.
         mtcars.shape[1]

Out[67]: 12

In [68]: # please note that the first column of mtcars is the "brand" column, not one containing indices
         mtcars.iloc[:,0]

Out[68]: 0               Mazda RX4
         1           Mazda RX4 Wag
         2              Datsun 710
         3          Hornet 4 Drive
         4       Hornet Sportabout
         5                 Valiant
         6              Duster 360
         7               Merc 240D
         8                Merc 230
         9                Merc 280
         10              Merc 280C
         11             Merc 450SE
         12             Merc 450SL
         13            Merc 450SLC
         14     Cadillac Fleetwood
         15    Lincoln Continental
         16      Chrysler Imperial
         17               Fiat 128
         18            Honda Civic
         19         Toyota Corolla
         20          Toyota Corona
         21       Dodge Challenger
         22            AMC Javelin
```

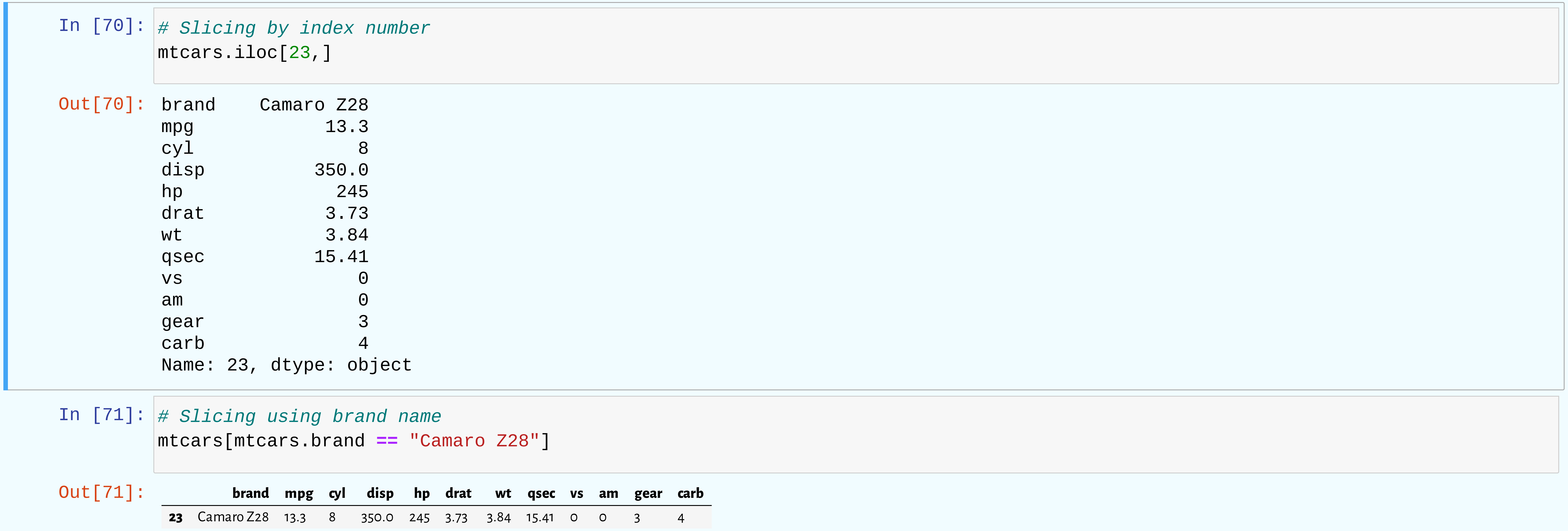


In [69]:

```
# display the structure of a dataframe
mtcars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   brand   32 non-null     object 
 1   mpg     32 non-null     float64
 2   cyl     32 non-null     int64  
 3   disp    32 non-null     float64
 4   hp      32 non-null     int64  
 5   drat    32 non-null     float64
 6   wt      32 non-null     float64
 7   qsec    32 non-null     float64
 8   vs      32 non-null     int64  
 9   am      32 non-null     int64  
 10  gear    32 non-null     int64  
 11  carb    32 non-null     int64  
dtypes: float64(5), int64(6), object(1)
memory usage: 3.1+ KB
```

SLICING DATA FRAMES BY ROW



```
In [70]: # Slicing by index number
         mtcars.iloc[23,]

Out[70]: brand    Camaro Z28
         mpg            13.3
         cyl               8
         disp          350.0
         hp              245
         drat           3.73
         wt             3.84
         qsec          15.41
         vs                0
         am                0
         gear              3
         carb              4
         Name: 23, dtype: object
```

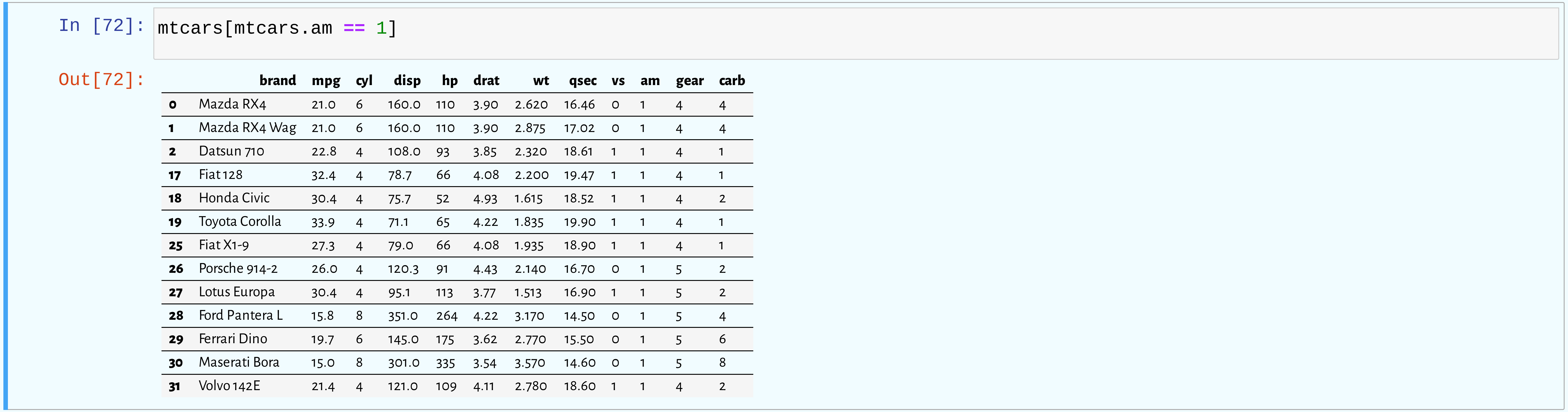
```
In [71]: # Slicing using brand name
         mtcars[mtcars.brand == "Camaro Z28"]
```

Out[71]:

	brand	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
23	Camaro Z28	13.3	8	350.0	245	3.73	3.84	15.41	0	0	3	4

SLICING DATA FRAMES BY ROW

Index by **logical expression**, for instance all cars with automatic transmission.



In [72]:

```
mtcars[mtcars.am == 1]
```

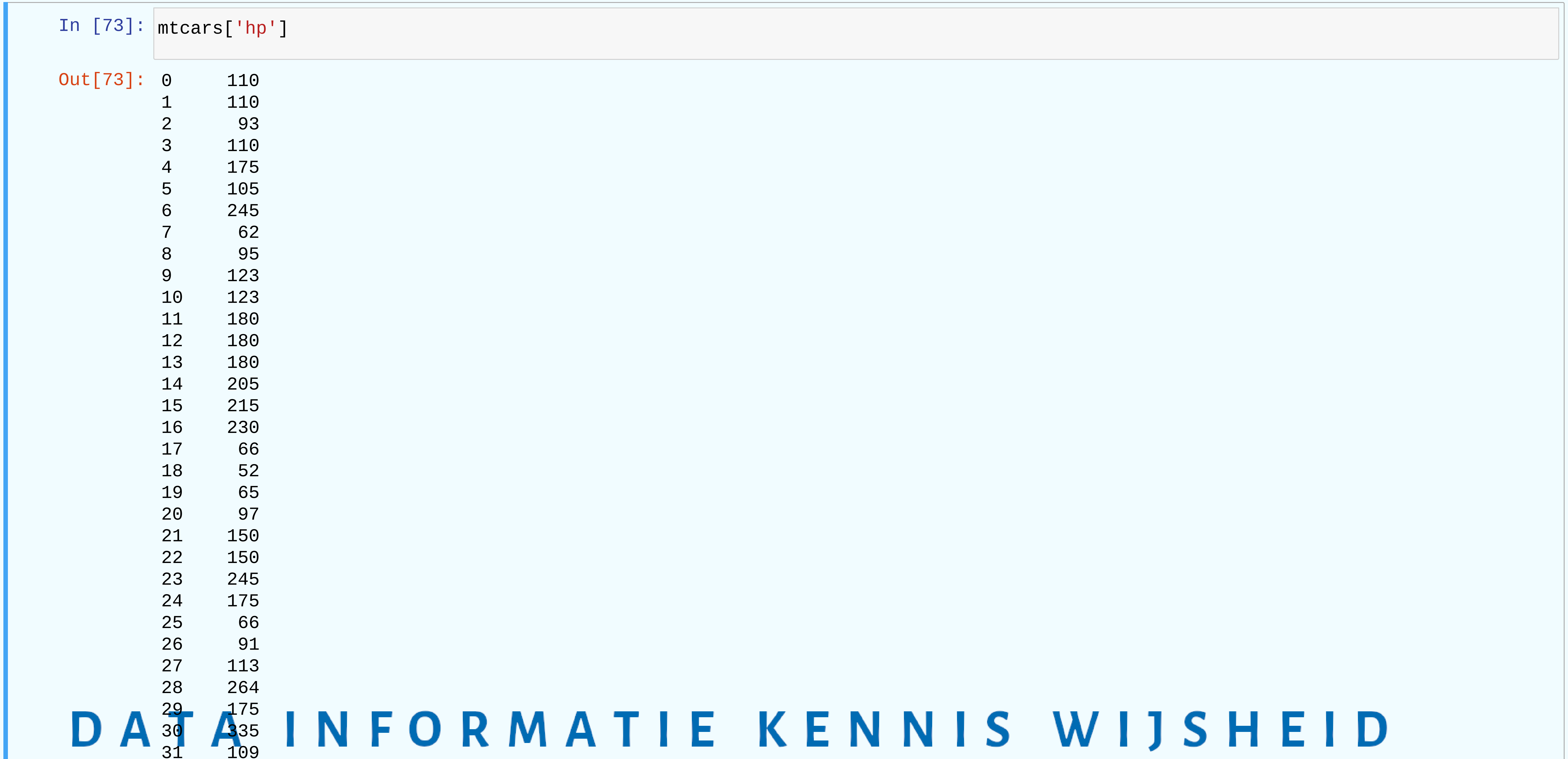
Out[72]:

	brand	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
17	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
18	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
19	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
25	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
26	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
27	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
28	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
29	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
30	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
31	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Other logical expressions: <, >, <=, >=, !=, |, &. **Try them!**

SLICING DATA FRAMES BY COLUMN

To select only one column, use square brackets



In [73]:

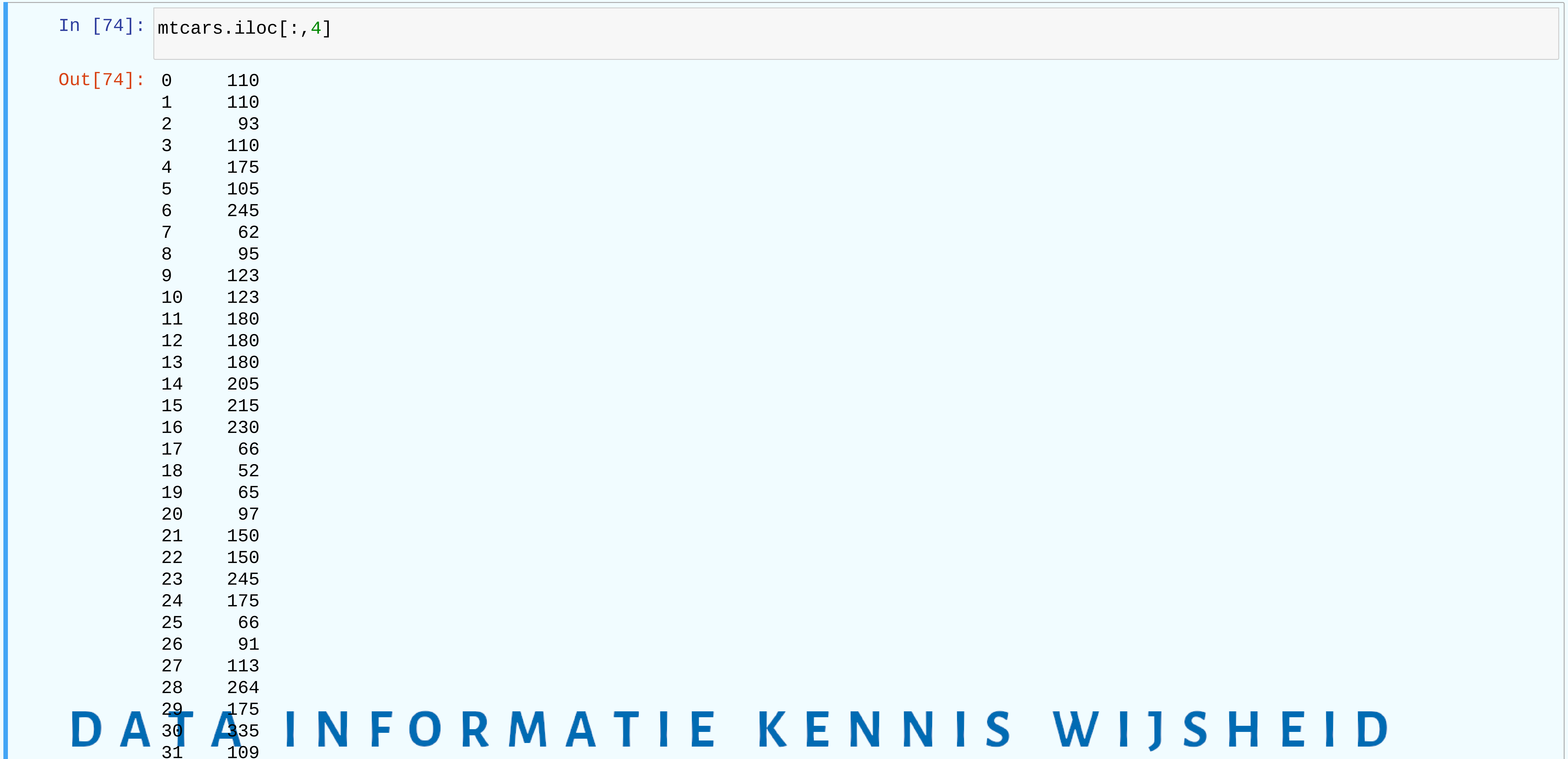
```
mtcars['hp']
```

Out[73]:

```
0      110
1      110
2       93
3      110
4      175
5      105
6      245
7       62
8       95
9      123
10     123
11     180
12     180
13     180
14     205
15     215
16     230
17      66
18      52
19      65
20      97
21     150
22     150
23     245
24     175
25      66
26      91
27     113
28     264
29     175
30     335
31     109
```

SLICING DATA FRAMES BY COLUMN

The same can be achieved by using the index for column hp. Since this is the fifth column, the corresponding index is 4.



```
In [74]: mtcars.iloc[:,4]
```

```
Out[74]: 0     110
         1     110
         2      93
         3     110
         4     175
         5     105
         6     245
         7      62
         8      95
         9     123
         10    123
         11    180
         12    180
         13    180
         14    205
         15    215
         16    230
         17     66
         18     52
         19     65
         20     97
         21    150
         22    150
         23    245
         24    175
         25     66
         26     91
         27    113
         28    264
         29    175
         30    335
         31    109
```

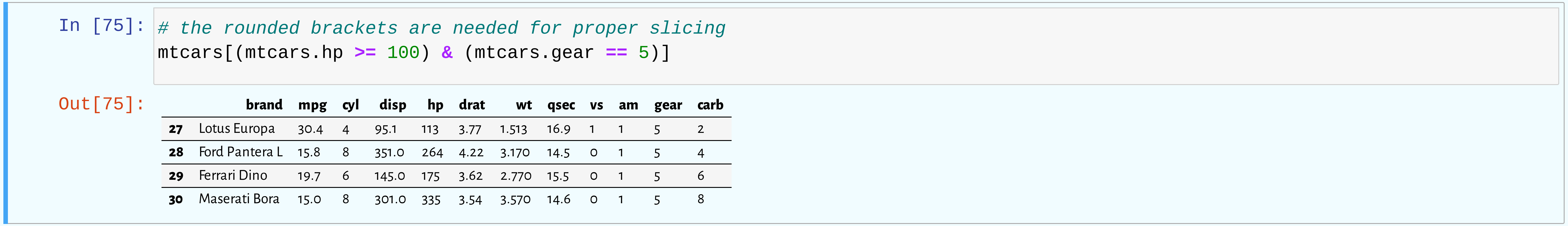
DIKW
ACADEMY

EXERCISE: SLICING DATA FRAMES BY ROW

Select only rows of cars with 100+ horsepower and 5 gears

SOLUTION: SLICING DATA FRAMES BY ROW

Select only rows of cars with 100+ horsepower and 5 gears



In [75]:

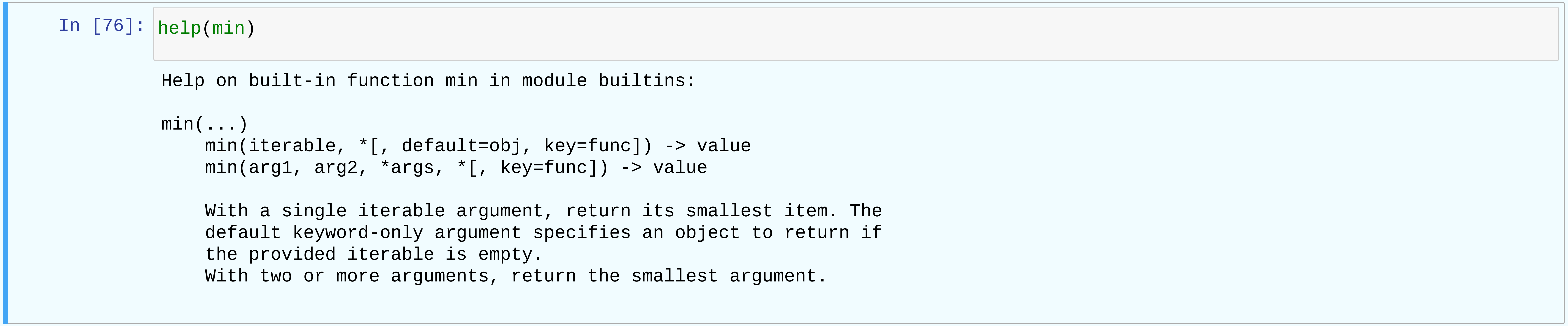
```
# the rounded brackets are needed for proper slicing
mtcars[(mtcars.hp >= 100) & (mtcars.gear == 5)]
```

Out[75]:

	brand	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
27	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.9	1	1	5	2
28	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.5	0	1	5	4
29	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.5	0	1	5	6
30	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.6	0	1	5	8

HELP FUNCTION IN PYTHON

Python's help() function invokes the interactive built-in help system



```
In [76]: help(min)
```

```
Help on built-in function min in module builtins:

min(...)
    min(iterable, *[, default=obj, key=func]) -> value
    min(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its smallest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the smallest argument.
```

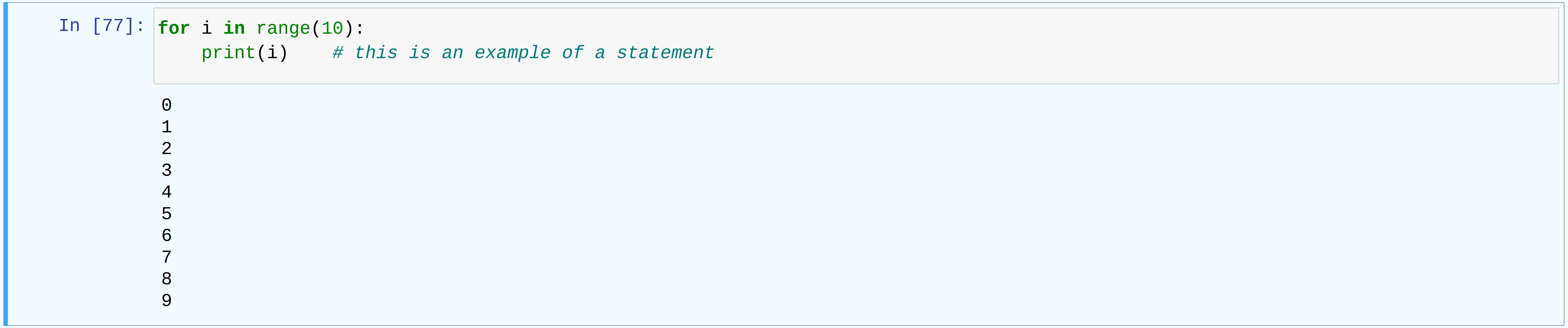
LOOPS AND FUNCTIONS

- For loops
- Repeat, break
- Basic functions
- Create your own functions

FOR LOOP

Use a For-loop when the number of iterations is predefined. Also be aware of the indentation.

Syntax in Python:



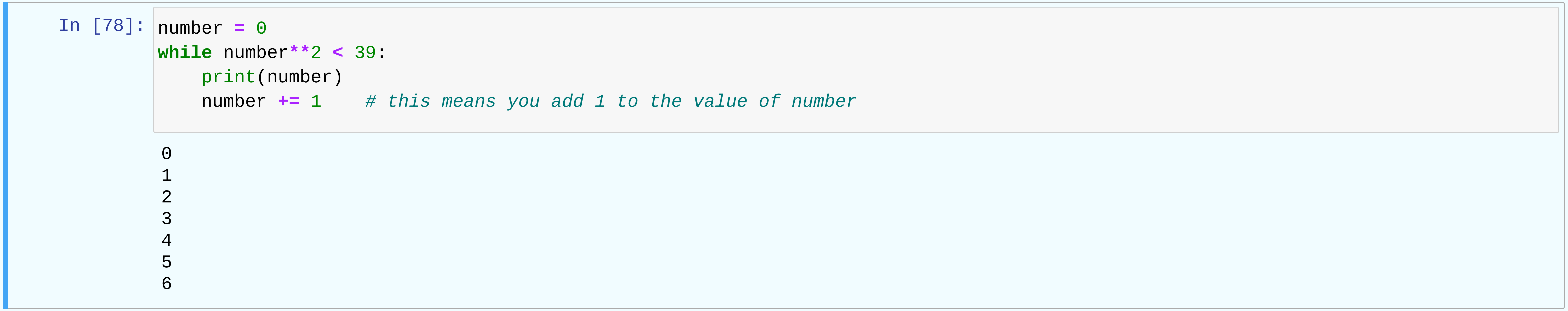
In [77]:

```
for i in range(10):
    print(i)    # this is an example of a statement
```

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

WHILE LOOP

Use a While-loop when the number of iterations is not predefined

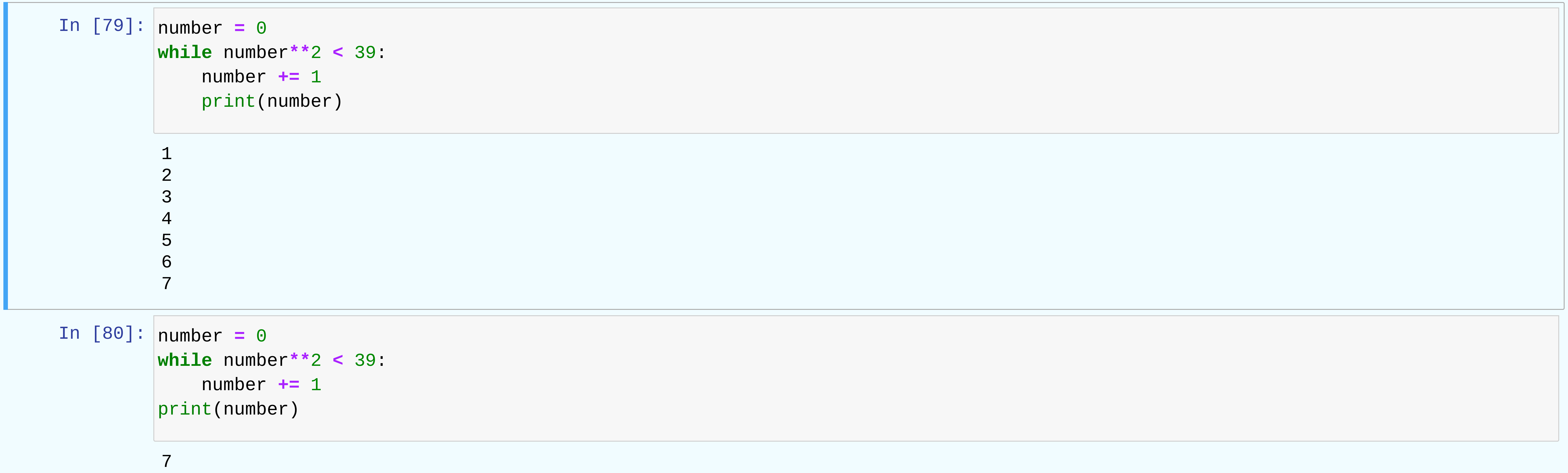


In [78]:

```
number = 0
while number**2 < 39:
    print(number)
    number += 1     # this means you add 1 to the value of number
```

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

Also note that the print-statement must be within the loop if you like to print a number during every iteration. Compare the following syntax.



```
In [79]: number = 0
         while number**2 < 39:
             number += 1
             print(number)

1
2
3
4
5
6
7
```

```
In [80]: number = 0
         while number**2 < 39:
             number += 1
         print(number)

7
```

FUNCTIONS

There are numerous basic built-in functions implemented in Python. You can even get more functions by installing more Python libraries.

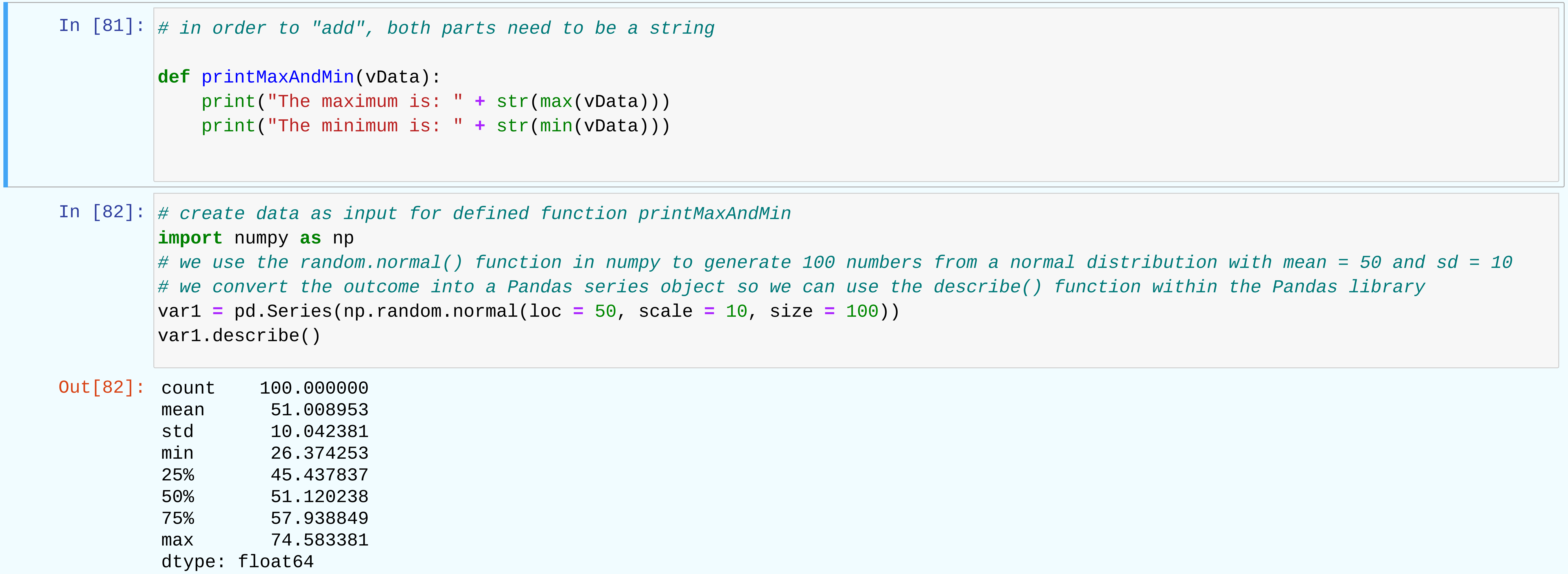
Some built-in functions we have seen: max(), print(), range().

A list of built-in functions can be found here

FUNCTIONS

It is also possible in Python to create your own function.

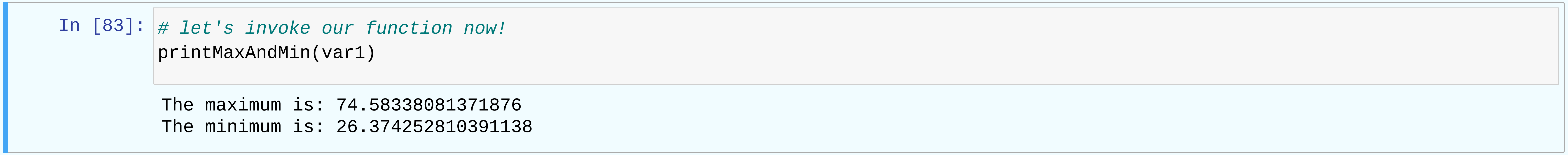
First you need to define your function (beware of the indentation!):



```
In [81]: # in order to "add", both parts need to be a string

         def printMaxAndMin(vData):
             print("The maximum is: " + str(max(vData)))
             print("The minimum is: " + str(min(vData)))
```

```
In [82]: # create data as input for defined function printMaxAndMin
         import numpy as np
         # we use the random.normal() function in numpy to generate 100 numbers from a normal distribution with mean = 50 and sd = 10
         # we convert the outcome into a Pandas series object so we can use the describe() function within the Pandas library
         var1 = pd.Series(np.random.normal(loc = 50, scale = 10, size = 100))
         var1.describe()
```

```
Out[82]: count    100.000000
         mean      51.008953
         std       10.042381
         min       26.374253
         25%       45.437837
         50%       51.120238
         75%       57.938849
         max       74.583381
         dtype: float64
```



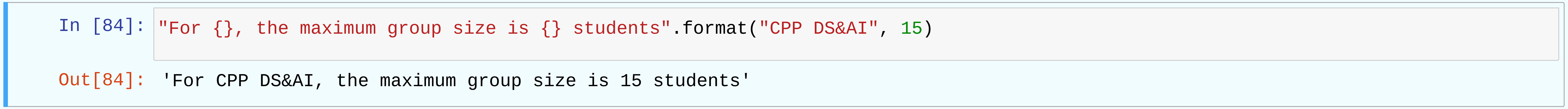
In [83]:

```
# let's invoke our function now!
printMaxAndMin(var1)
```

```
The maximum is: 74.58338081371876
The minimum is: 26.374252810391138
```

STRING FORMAT METHOD

With this method, you can insert values in string placeholders



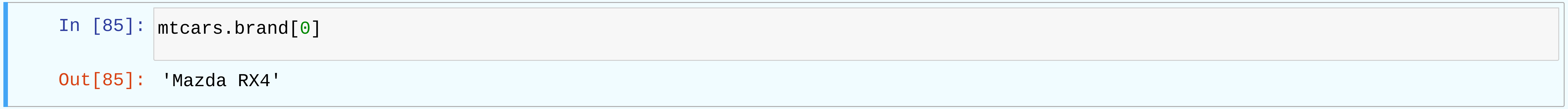
```
In [84]: "For {}, the maximum group size is {} students".format("CPP DS&AI", 15)

Out[84]: 'For CPP DS&AI, the maximum group size is 15 students'
```

EXERCISE FUNCTIONS

Now create your own function! Make a function that prints the mpg and hp for a given car in the mtcars dataset.

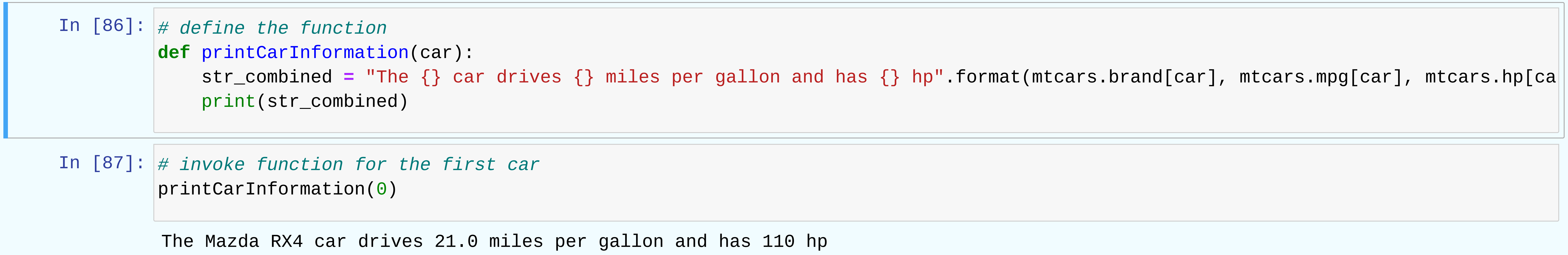
Advanced: Then use a for-statement to print the information for all cars.

Tip: use mtcars.brand[0] to get the first carname



```
In [85]: mtcars.brand[0]

Out[85]: 'Mazda RX4'
```

SOLUTION FUNCTIONS (I)

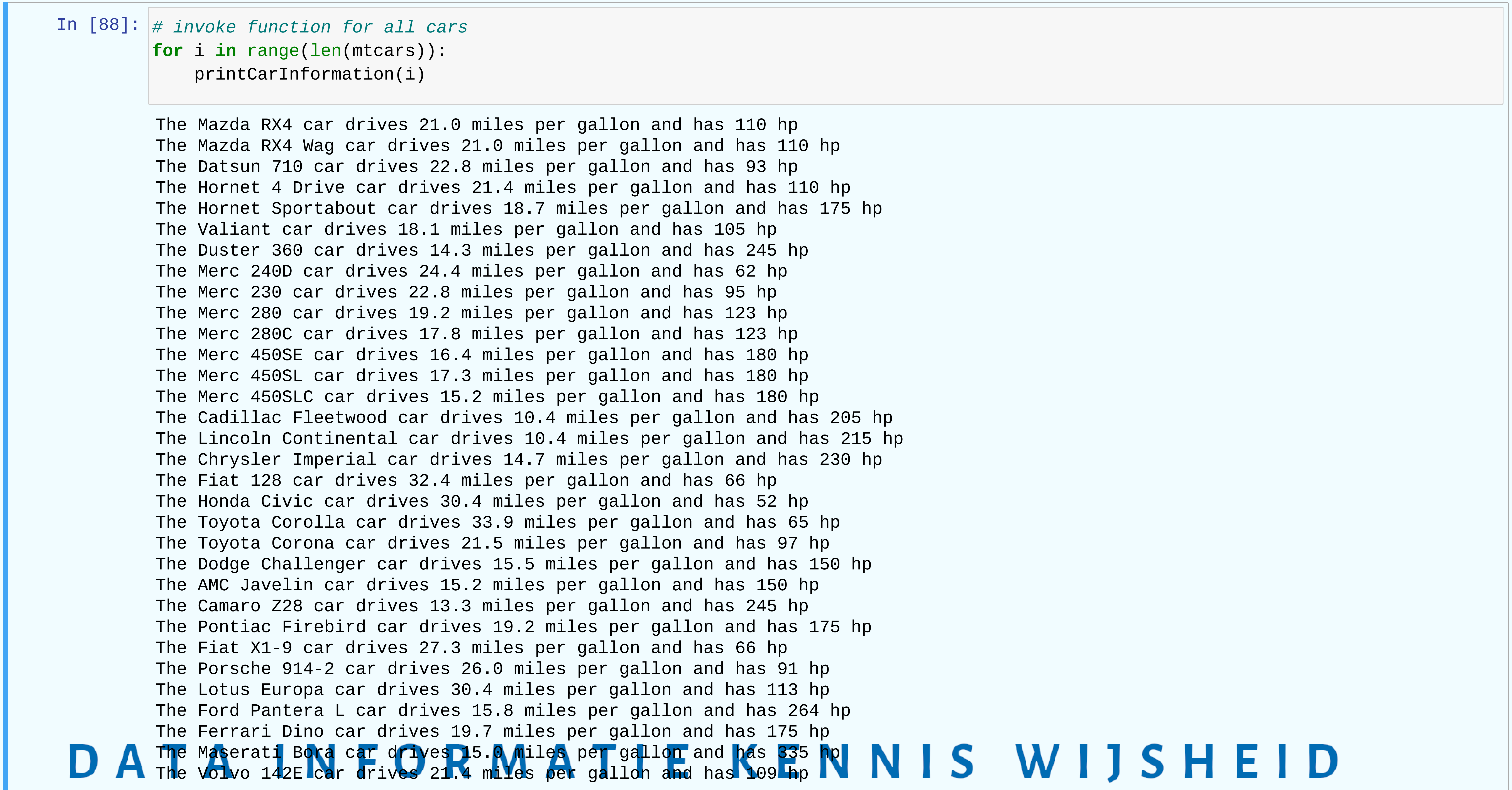


```
In [86]: # define the function
         def printCarInformation(car):
             str_combined = "The {} car drives {} miles per gallon and has {} hp".format(mtcars.brand[car], mtcars.mpg[car], mtcars.hp[ca
             print(str_combined)
```

```
In [87]: # invoke function for the first car
         printCarInformation(0)
```

```
The Mazda RX4 car drives 21.0 miles per gallon and has 110 hp
```

SOLUTION FUNCTIONS (II)



```
In [88]: # invoke function for all cars
         for i in range(len(mtcars)):
             printCarInformation(i)
```

```
The Mazda RX4 car drives 21.0 miles per gallon and has 110 hp
The Mazda RX4 Wag car drives 21.0 miles per gallon and has 110 hp
The Datsun 710 car drives 22.8 miles per gallon and has 93 hp
The Hornet 4 Drive car drives 21.4 miles per gallon and has 110 hp
The Hornet Sportabout car drives 18.7 miles per gallon and has 175 hp
The Valiant car drives 18.1 miles per gallon and has 105 hp
The Duster 360 car drives 14.3 miles per gallon and has 245 hp
The Merc 240D car drives 24.4 miles per gallon and has 62 hp
The Merc 230 car drives 22.8 miles per gallon and has 95 hp
The Merc 280 car drives 19.2 miles per gallon and has 123 hp
The Merc 280C car drives 17.8 miles per gallon and has 123 hp
The Merc 450SE car drives 16.4 miles per gallon and has 180 hp
The Merc 450SL car drives 17.3 miles per gallon and has 180 hp
The Merc 450SLC car drives 15.2 miles per gallon and has 180 hp
The Cadillac Fleetwood car drives 10.4 miles per gallon and has 205 hp
The Lincoln Continental car drives 10.4 miles per gallon and has 215 hp
The Chrysler Imperial car drives 14.7 miles per gallon and has 230 hp
The Fiat 128 car drives 32.4 miles per gallon and has 66 hp
The Honda Civic car drives 30.4 miles per gallon and has 52 hp
The Toyota Corolla car drives 33.9 miles per gallon and has 65 hp
The Toyota Corona car drives 21.5 miles per gallon and has 97 hp
The Dodge Challenger car drives 15.5 miles per gallon and has 150 hp
The AMC Javelin car drives 15.2 miles per gallon and has 150 hp
The Camaro Z28 car drives 13.3 miles per gallon and has 245 hp
The Pontiac Firebird car drives 19.2 miles per gallon and has 175 hp
The Fiat X1-9 car drives 27.3 miles per gallon and has 66 hp
The Porsche 914-2 car drives 26.0 miles per gallon and has 91 hp
The Lotus Europa car drives 30.4 miles per gallon and has 113 hp
The Ford Pantera L car drives 15.8 miles per gallon and has 264 hp
The Ferrari Dino car drives 19.7 miles per gallon and has 175 hp
The Maserati Bora car drives 15.0 miles per gallon and has 335 hp
The Volvo 142E car drives 21.4 miles per gallon and has 109 hp
```