Machine Learning for Economics and Finance

Python Intro

Ole Wilms

October 17, 2024

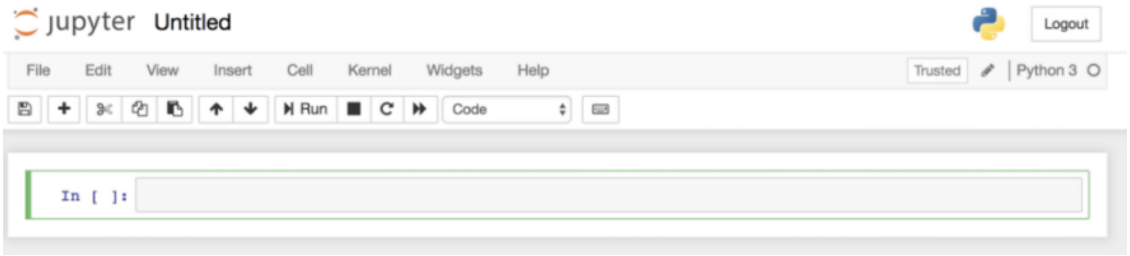
Contents

Important Instructions

- The purpose of this introduction is to get to know Python by some basic programming exercises.
- In case you struggle with some problems, please post your questions on the OpenOlat Forum.
- Don't worry if you struggle at the beginning. Throughout the course, these programming concepts will become easier to understand.
- Consistent practice with basic Python tasks is key to mastering the language. I strongly encourage all students to engage in regular exercises and to proactively tackle future tasks to strengthen your skills and build confidence in your abilities.

Recommended integrated development environments (IDE's) for Python

Jupyter (notebooks) - Recommended for this lecture



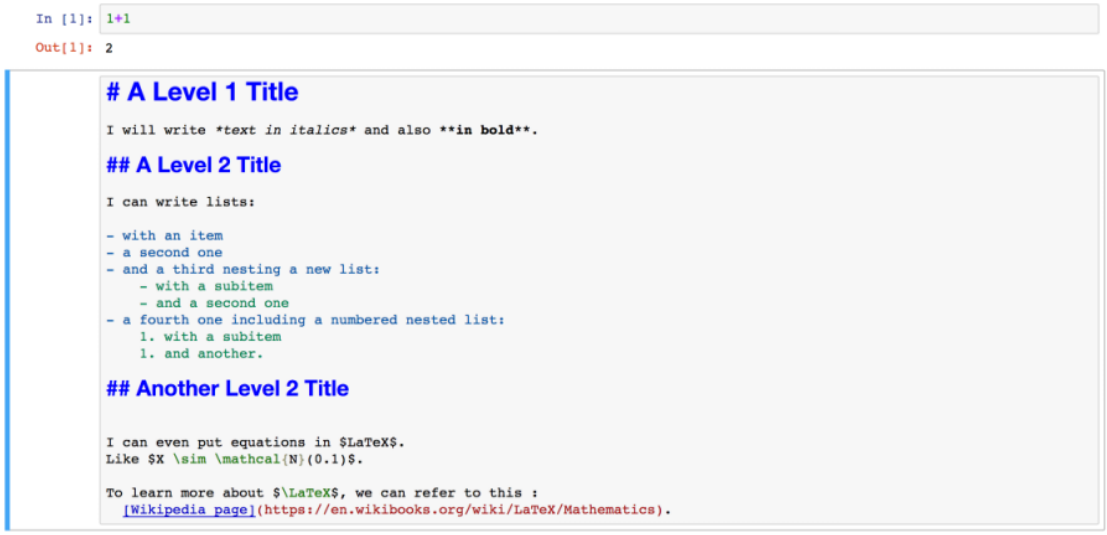
Jupyterlab direct website-IDE try it out!

or

Jupyter IDE weblink for installation.

Jupyter is a graphical user interface in a web browser. It is an open-source application for creating and sharing documents that contain code, equations, graphical representations and text. It is possible to include and execute different language codes in Jupyter notebooks (e.g. **Code** cells like *Python*, **Markdown** cells or even raw cells).

Markdown cell (syntax example):



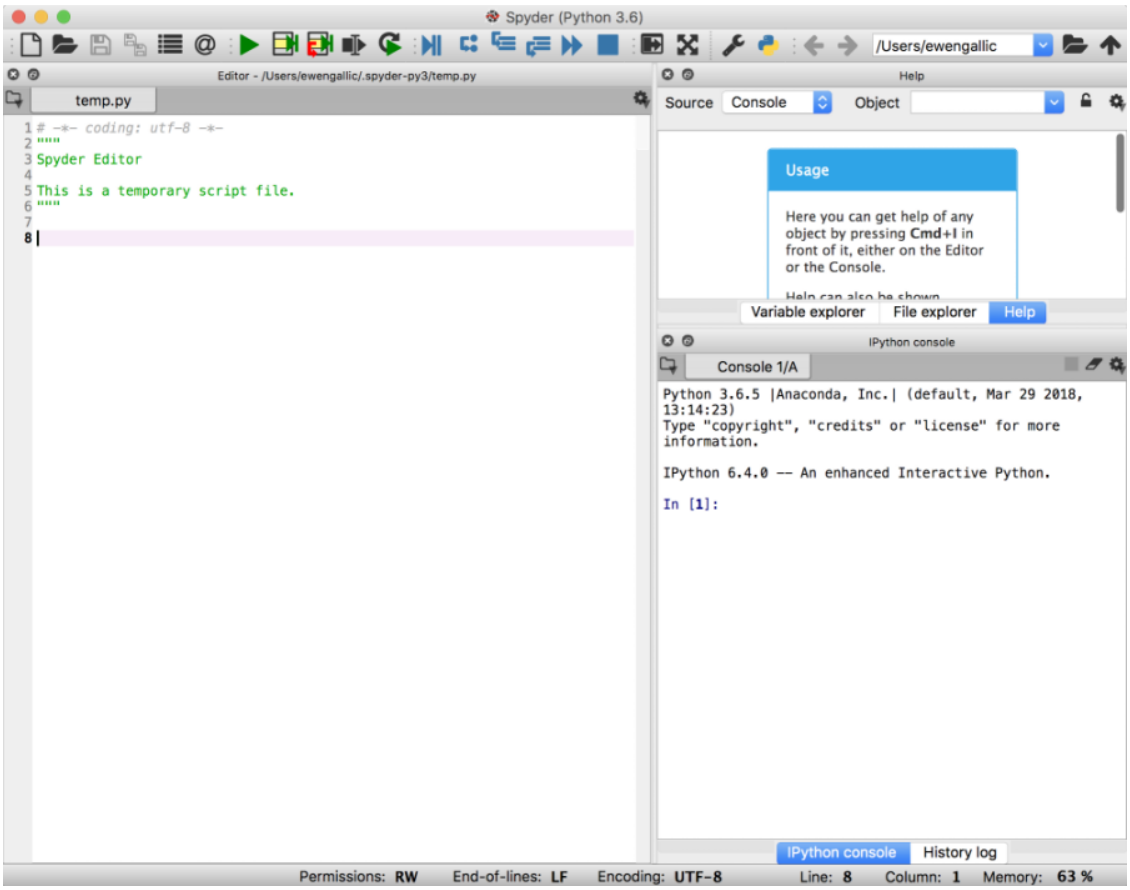
```
In [1]: 1+1
Out[1]: 2
```

```
# A Level 1 Title

I will write *text in italics* and also **in bold**.

## A Level 2 Title

I can write lists:

- with an item
- a second one
- and a third nesting a new list:
    - with a subitem
    - and a second one
- a fourth one including a numbered nested list:
    1. with a subitem
    1. and another.

## Another Level 2 Title


I can even put equations in $LaTeX$.
Like $X \sim \mathcal{N}(0.1)$.

To learn more about $\LaTeX$, we can refer to this :
  [Wikipedia page](https://en.wikibooks.org/wiki/LaTeX/Mathematics).
```

Note

Jupyter Notebook can be launched by Jupyter Desktop or via terminal. While executing the **.ipynb** file, the default web browser launches and a server will be launched as well as a Python process (a kernel). If the browser does not launch automatically, the page that should have been displayed can be accessed at the following address: `http://localhost:8890/tree?`.

Spyder



Spyder IDE weblink for installation.

Spyder is a single integrated development environment (IDE) that includes both an editor and a console. This is what Spyder offers, with many additional features, such as project management, variable inspector, file explorer, command log, debugger, etc.

Exercise 1: Getting started

Defining variables

[1]:
```
x = 5
y = 7
print(x)
```

```
5
```

[2]:
```
z = x * y
print(z)
```

```
35
```

[3]:
```
a = x**y
print(a)
```

```
78125
```

Jupyter Extensions and Python packages

Jupyter Extensions (ipywidget recommendation)

Table of Contents extension:

- Extension name: `@jupyterlab/toc-extension`

Jupyter extension to display the table of contents.

Variable Inspector:

- Extension name: `@lckr/jupyterlab_variableinspector`

When you use the *variableinspector* it will allow you to get a side-by-side overview of your variables besides your code.

Python packages

Installing missing packages

[4]:
```
%%script echo skipping
python -m pip install "package name"
```

```
skipping
```

Loading packages

[5]:
```
%%script echo skipping
import "package name"
```

```
skipping
```

Some basic functions in Python are loaded by default. Others require a module to be loaded. These **modules** are files that contain *definitions* as well as *instructions*. **Package** are defined as a combination of modules that offer a set of functions. Among the packages that will be used in these notes are:

- `NumPy`, a fundamental package for scientific calculations
- `pandas`, a package allowing easy data manipulation and analysis
- `Matplotlib`, a package allowing us to create graphics.

To load a module (or a package), we use the command import. For example, to load the package `numpy`:

[6]:
```
import numpy as np

b = np.log(a)  # here we use the numpy package to call the function log()
print(b)
```

```
11.266065387038703
```

[7]:
```
# remove x
del(x)
```

[8]:
```
#dir()  # show all objects
all_variables = dir()

# Iterate over the whole list where dir( ) is stored.
for myvariable in all_variables:
    # Print the item if it doesn't start with '_'
    if not myvariable.startswith('_'):
        myvalue = eval(myvariable)
        print(myvariable, "is", type(myvalue))
```

```
In is <class 'list'>
NamespaceMagics is <class 'traitlets.traitlets.MetaHasTraits'>
Out is <class 'dict'>
a is <class 'int'>
b is <class 'numpy.float64'>
exit is <class 'IPython.core.autocall.ZMQExitAutocall'>
get_ipython is <class 'function'>
json is <class 'module'>
np is <class 'module'>
open is <class 'function'>
quit is <class 'IPython.core.autocall.ZMQExitAutocall'>
sys is <class 'module'>
y is <class 'int'>
z is <class 'int'>
```

The Help System

The help can be accessed using different syntaxes:

- `?` : provides an introduction and an overview of the features offered in Python (you leave it with the *ESC* key)
- `object?` : provides details about object (for example `x?` or `plt.plot?`)
- `object??` : more details about object
- `%quickref` : short reference on Python syntaxes
- `help()` : access to the Python help system.

Note: the tabulation key on the keyboard allows not only *autocompletion*, but also an *exploration of the content* of an object or module.

In addition, when it comes to finding help on a more complex problem, the right thing to do is not hesitate to search on a search engine, in mailing lists and of course on the many questions on Stack Overflow .

Working with vectors

[9]:
```
x = [1, 4, 5]
print(x)
```

```
[1, 4, 5]
```

[10]:
```
sum(x)  # sum all elements
```

[10]:
```
10
```

[11]:
```
# Same as the help(sum) function
?sum
```

```
Signature: sum(iterable, /, start=0)
Docstring:
Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value.
This function is intended specifically for use with numeric values and may
reject non-numeric types.
Type:      builtin_function_or_method
```

[12]:
```
np.mean(x)  # compute mean
```

[12]:
```
3.3333333333333335
```

[13]:
```
sd_x = np.std(x)  # compute standard deviation and store in sd_x
print(sd_x)
```

```
1.699673171197595
```

[14]:
```
y = [2, 3, 1]
print(y)
```

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

[15]:
```
z = x + y
print(z)
```

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

[16]:
```
#z = x * y  # Element by element multiplication
# Multiplying list 'a' elementwise with list 'b':
z = list(map(lambda a, b: a * b, x, y))
print(z)
```

```
[2, 12, 5]
```

[17]:
```
len(z)
```

[17]: 3

Special "Numbers"

[18]:
```
# Example 1:
# 1/0   <--- This is not allowed since it represents a division by zero   #␣
 ↪Infinity
# instead:
print(float('inf'))
print(float('-inf'))

1/float('inf')
```

```
inf
-inf
```

[18]: 0.0

Types of Data class types and vector modes

In Python, there are several class types that are used to represent different types of data objects. Some of the commonly used class types include: *Integer* (Numeric), *Float*, *String* (Character), *Factor* and *Logical*.

Numeric: Represents integer (real) values

[19]:
```
x = [1, 4, 5]
print(type(x))
```

```
<class 'list'>
```

Logical: Represents logical (Boolean) values `True` or `False`

[20]:
```
# Check if x is a vector of integers
all(isinstance(i, (int)) for i in x)
```

[20]:
```
True
```

Float: Represents floating Point Numbers

[21]:
```
x2 = [1.2, 2, 3]

# Check if x2 is a vector of integers
all(isinstance(i, (int)) for i in x2)
```

[21]:
```
False
```

[22]:
```
import numpy as np

x3 = np.array([1, 4, 5])  # Example array
print(x3.dtype)  # getting the data type of the array
```

```
int64
```

[23]:
```
# Check if x2 is a vector of floats
all(isinstance(i, (float)) for i in x2)
```

[23]:
```
False
```

[24]:
```
# Check if x2 is a vector of integers and floats
all(isinstance(i, (int, float)) for i in x2)
```

[24]:
```
True
```

Conversions

[25]:
```
x4 = "3"
x_int = int(x4)
print(type(x_int))
```

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

`int()`, `float()`, and `str()` convert values to integers, floats, and strings. Example: Converting a float to an integer can be done with `int(1.2)`, which results in 1.

[26]:
```
x2 = [int(i) for i in x2]  # Convert all elements in x2 to integers
print(x2)
```

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

[27]:
```
# Check again if x2 is a vector of ints
all(isinstance(i, (int)) for i in x2)
```

[27]:
```
True
```

String: Represents characters (text) values

[28]:
```
t = "Hello World"  # single line text
print(t)
```

```
Hello World
```

[29]:
```
t2 = """Hello \
World"""   # multi line text
print(t2)
```

```
Hello World
```

Factor: Represents categorical variables with a fixed number of levels

[30]:
```
import pandas as pd

gender = pd.DataFrame(['Male', 'Female', 'Diverse', 'Male'])

print('Unique entries: ', pd.Series({c: gender[c].unique() for c in gender}))
```

```
Unique entries:  0    [Male, Female, Diverse]
dtype: object
```

[31]:
```
import pandas as pd
g = pd.Series(['Male', 'Female', 'Diverse', 'Male'])
cat_s = g.astype('category')

# Checking the data types and unique entries (levels)
cat_s

# Alternatively:
# cat_s.cat.categories
```

[31]:
```
0       Male
1     Female
2    Diverse
3       Male
dtype: category
Categories (3, object): ['Diverse', 'Female', 'Male']
```

[32]:
```
#cat_s.cat.codes

actual_categories = ['Male', 'Female', 'Diverse', 'Other']
```

```
cat_s2 = cat_s.cat.set_categories(actual_categories)
cat_s2
```

[32]:
```
0       Male
1     Female
2    Diverse
3       Male
dtype: category
Categories (4, object): ['Male', 'Female', 'Diverse', 'Other']
```

Difference between an none and NaN

[33]:
```
import numpy as np

a = None  # setting a to an empty object (placeholder)
b = np.nan  # setting 'b' to 'NaN'
# b = float("nan")  # equal to the line above

print(np.isnan(b))  # checking if 'a' equals 'NaN'
```

```
True
```

[34]:
```
print(a)
```

```
None
```

[35]:
```
print(type(a))  # The None object is a neutral variable, with "null" behavior␣
↪(an empty placeholder).
```

```
<class 'NoneType'>
```

[36]:
```
print(type(b))  # Representation as a floating point number of 'Not a Number'.
```

```
<class 'float'>
```

Exercise 2: Loading and changing Data

Get / Set working directory path

Loading datasets from package

[37]:
```
from ISLP import load_data

# Load Hitters dataset from the ISLP package
Hitters = load_data('Hitters')
```

Loading datasets from external files

Loading data from a CSV file

[38]:
```
%%script echo skipping
import pandas as pd
df = pd.read_csv("PATH/filename.csv",
                 sep=";",
                 index_col=0,
                 decimal=",")
print(df.head())
```

```
skipping
```

Creating DataFrame

Group by example

[39]:
```
import pandas as pd
import numpy as np

# Example data
unemployment = pd.DataFrame({
    "departement" : ["R&D", "I.T.",
                     "Accounting", "Marketing",
                     "Sales", "Finance"]*2,
    "workers" : [8738, 12701, 11390, 10228, 975, 1297,
                 8113, 12258, 10897, 9617, 936, 1220],
    "engineers" : [1420, 2530, 3986, 2025, 259, 254,
                   1334, 2401, 3776, 1979, 253, 241]
    })

# unemployed workers (thereof engineers) by department
unemployment.loc[:,["departement", "workers", "engineers"]].
 ↪groupby("departement").sum()
```

[39]:
```
             workers  engineers
departement
```

```
Accounting      22287       7762
Finance          2517        495
I.T.            24959       4931
Marketing       19845       4004
R&D             16851       2754
Sales            1911        512
```

Data inspection

[40]:
```
Hitters.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 322 entries, 0 to 321
Data columns (total 20 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   AtBat      322 non-null    int64
 1   Hits       322 non-null    int64
 2   HmRun      322 non-null    int64
 3   Runs       322 non-null    int64
 4   RBI        322 non-null    int64
 5   Walks      322 non-null    int64
 6   Years      322 non-null    int64
 7   CAtBat     322 non-null    int64
 8   CHits      322 non-null    int64
 9   CHmRun     322 non-null    int64
 10  CRuns      322 non-null    int64
 11  CRBI       322 non-null    int64
 12  CWalks     322 non-null    int64
 13  League     322 non-null    category
 14  Division   322 non-null    category
 15  PutOuts    322 non-null    int64
 16  Assists    322 non-null    int64
 17  Errors     322 non-null    int64
 18  Salary     263 non-null    float64
 19  NewLeague  322 non-null    category
dtypes: category(3), float64(1), int64(16)
memory usage: 44.2 KB
```

[41]:
```
# Show the first five rows - with all columns
Hitters.head(5)
```

[41]:

	AtBat	Hits	HmRun	Runs	RBI	Walks	Years	CAtBat	CHits	CHmRun	CRuns	\
0	293	66	1	30	29	14	1	293	66	1	30	
1	315	81	7	24	38	39	14	3449	835	69	321	
2	479	130	18	66	72	76	3	1624	457	63	224	
3	496	141	20	65	78	37	11	5628	1575	225	828	
4	321	87	10	39	42	30	2	396	101	12	48	

```
   CRBI  CWalks League Division  PutOuts  Assists  Errors  Salary NewLeague
0    29      14      A        E      446       33      20     NaN          A
1   414     375      N        W      632       43      10   475.0          N
2   266     263      A        W      880       82      14   480.0          A
3   838     354      N        E      200       11       3   500.0          N
4    46      33      N        E      805       40       4    91.5          N
```

[42]:
```
# Show the first five rows - with all columns
Hitters.iloc[[0, 1, 2, 3, 4]]   # iloc[[ROW][COLUMN]]
```

[42]:
```
   AtBat  Hits  HmRun  Runs  RBI  Walks  Years  CAtBat  CHits  CHmRun  CRuns  \
0    293    66      1    30   29     14      1     293     66       1     30
1    315    81      7    24   38     39     14    3449    835      69    321
2    479   130     18    66   72     76      3    1624    457      63    224
3    496   141     20    65   78     37     11    5628   1575     225    828
4    321    87     10    39   42     30      2     396    101      12     48

   CRBI  CWalks League Division  PutOuts  Assists  Errors  Salary NewLeague
0    29      14      A        E      446       33      20     NaN          A
1   414     375      N        W      632       43      10   475.0          N
2   266     263      A        W      880       82      14   480.0          A
3   838     354      N        E      200       11       3   500.0          N
4    46      33      N        E      805       40       4    91.5          N
```

[43]:
```
# Show the first five rows - with all columns
Hitters.iloc[0:5,:]
```

[43]:
```
   AtBat  Hits  HmRun  Runs  RBI  Walks  Years  CAtBat  CHits  CHmRun  CRuns  \
0    293    66      1    30   29     14      1     293     66       1     30
1    315    81      7    24   38     39     14    3449    835      69    321
2    479   130     18    66   72     76      3    1624    457      63    224
3    496   141     20    65   78     37     11    5628   1575     225    828
4    321    87     10    39   42     30      2     396    101      12     48

   CRBI  CWalks League Division  PutOuts  Assists  Errors  Salary NewLeague
0    29      14      A        E      446       33      20     NaN          A
1   414     375      N        W      632       43      10   475.0          N
2   266     263      A        W      880       82      14   480.0          A
3   838     354      N        E      200       11       3   500.0          N
4    46      33      N        E      805       40       4    91.5          N
```

[44]:
```
Hitters.describe().T  #Descriptive table - wherein ".T" transposes the table
```

[44]:
```
          count         mean          std   min     25%     50%      75%  \
AtBat     322.0   380.928571   153.404981  16.0  255.25   379.5   512.00
Hits      322.0   101.024845    46.454741   1.0   64.00    96.0   137.00
```

```
HmRun     322.0    10.770186     8.709037   0.0    4.00     8.0    16.00
Runs      322.0    50.909938    26.024095   0.0   30.25    48.0    69.00
RBI       322.0    48.027950    26.166895   0.0   28.00    44.0    64.75
Walks     322.0    38.742236    21.639327   0.0   22.00    35.0    53.00
Years     322.0     7.444099     4.926087   1.0    4.00     6.0    11.00
CAtBat    322.0  2648.683230  2324.205870  19.0  816.75  1928.0  3924.25
CHits     322.0   717.571429   654.472627   4.0  209.00   508.0  1059.25
CHmRun    322.0    69.490683    86.266061   0.0   14.00    37.5    90.00
CRuns     322.0   358.795031   334.105886   1.0  100.25   247.0   526.25
CRBI      322.0   330.118012   333.219617   0.0   88.75   220.5   426.25
CWalks    322.0   260.239130   267.058085   0.0   67.25   170.5   339.25
PutOuts   322.0   288.937888   280.704614   0.0  109.25   212.0   325.00
Assists   322.0   106.913043   136.854876   0.0    7.00    39.5   166.00
Errors    322.0     8.040373     6.368359   0.0    3.00     6.0    11.00
Salary    263.0   535.925882   451.118681  67.5  190.00   425.0   750.00
```

```
             max
AtBat      687.0
Hits       238.0
HmRun       40.0
Runs       130.0
RBI        121.0
Walks      105.0
Years       24.0
CAtBat   14053.0
CHits     4256.0
CHmRun     548.0
CRuns     2165.0
CRBI      1659.0
CWalks    1566.0
PutOuts   1378.0
Assists    492.0
Errors      32.0
Salary    2460.0
```

Missing data

[45]:
```
Hitters.isna().sum()
```

[45]:
```
AtBat        0
Hits         0
HmRun        0
Runs         0
RBI          0
Walks        0
Years        0
CAtBat       0
```

```
CHits          0
CHmRun         0
CRuns          0
CRBI           0
CWalks         0
League         0
Division       0
PutOuts        0
Assists        0
Errors         0
Salary        59
NewLeague      0
dtype: int64
```

[46]:
```
# Remove missing values
Hitters = Hitters.dropna()
```

Mixing data

Generating further example dataframes:

[47]:
```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    },
    index=[0, 1, 2, 3],
)

df2 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
    },
    index=[4, 5, 6, 7],
)

df3 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
```

```
    },
    index=[0, 1, 2, 3],
)
```

Concat

[48]:
```
df12_concat = pd.concat([df1,df2])
```

[49]:
```
df12_concat
```

[49]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

[50]:
```
df12_concat = pd.concat([df1,df3]) # Taking initial indexes into account -␣
↪hence index duplicates
```

[51]:
```
df12_concat
```

[51]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
0	A4	B4	C4	D4
1	A5	B5	C5	D5
2	A6	B6	C6	D6
3	A7	B7	C7	D7

[52]:
```
df12_concat = pd.concat([df1,df3], ignore_index=True) # Ignore initial indexes␣
↪- straight concat
```

[53]:
```
df12_concat
```

[53]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6

```
7  A7  B7  C7  D7
```

Join

[54]:
```
df12_join = pd.concat([df1,df2], axis=1)  # join='outer' !
```

[55]:
```
df12_join
```

[55]:

	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7

[56]:
```
df13_join = pd.concat([df1,df3], axis=1).reindex(df1.index) # Careful with this␣
↪one.
```

[57]:
```
df13_join
```

[57]:

	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	A4	B4	C4	D4
1	A1	B1	C1	D1	A5	B5	C5	D5
2	A2	B2	C2	D2	A6	B6	C6	D6
3	A3	B3	C3	D3	A7	B7	C7	D7

Exercise 3: Creating Basic Variables

Create Vector

[58]:
```
# Create a vector as a row
vector_row = np.array([1, 2, 3])

# Create a vector as a column
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Create Matrix

[59]:
```
ma = np.array([[1, 0],
               [0, 1]])

mb = np.array([[4, 1],
               [2, 2]])
```

[60]:
```
np.matmul(ma, mb)
```

[60]:
```
array([[4, 1],
       [2, 2]])
```

Generating sequences

[61]:
```
num_seq = range(5, 11)
len(num_seq)
```

[61]:
```
6
```

[62]:
```
for number in num_seq:
    print(number)
```

```
5
6
7
8
9
10
```

Simulate random variables

[63]:
```
import random
n_random = random.sample(range(1, 100), 3)
n_random
```

[63]:
```
[3, 76, 75]
```

Writing functions

[64]:
```
def my_function():
  print("Hello from a function")
```

[65]:
```
print(my_function())
```

```
Hello from a function
None
```

Exercise 4: Operators, control flow statements and loops

Comparison of boolean (logical) operators

In Python, boolean comparisons can only result in two logical values: `True` and `False`. These values represent the outcome of logical operations and comparisons in Python.

- `==` means "is equal".
  - The statement x == a framed as a question means: "Does the value of x equal the value of a?"
- `!=` means "not equal".
  - The statement x == b means: "Does the value of x not equal the value of b?"
- `<` means "less than".
  - The statement x < c means: "Is the value of x less than the value of c?"
- `<=` means "less than or equal".
  - The statement x <= d means: "Is the value of x less or equal to the value of d?"
- `>` means "greater than".
  - The statement x > e means: "Is the value of x greater than the value of e?"
- `>=` means "greater than or equal".
  - The statement x >= f means: "Is the value of x greater than or equal to the value of f?"

[66]:
```
#Example: Boolean operator comparison
x = 6
y = 5

x == y
```

[66]:
```
False
```

[67]:
```
output = x >= y

print(output)
```

```
True
```

Control flow statements - `if`, `elif` & `else`

[68]:
```
%%script echo skipping
# If-Else Statements: Test whether a certain condition is satisfied; then do␣
 ↪something

if condition:
    # body of if statement
else:
    # body of else statement
```

```
skipping
```

[69]:
```
if x > y:
    z = print("x is larger than y")
elif x == y:
    z = print("x is equal to y")
else:
    z = print("x is smaller than y")

z
```

```
x is larger than y
```

Loops - `for`

[70]:
```
x = []
n = 10

# Let's try to construct a vector that contains n entries: 1^2, 2^2, 3^2, 4^2,..
 ↪.
# Loops: repeat something i-times
for i in range(1, n+1):     # 1 till 10 equals 9 -> n+1
    x.append(i**2)

print(x)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[71]:
```
import numpy as np

# Shortcut:
x2 = np.arange(1, n+1)**2

# Alternativ:
#x2 = np.linspace(start=1, stop=n, num=n)**2

print(x2)
```

```
[  1   4   9  16  25  36  49  64  81 100]
```

[72]:
```
# Example: Use loop to find negative values in a vector
## Simulate random variables

np.random.seed(1)  # Fix the random seed for reproducibility

x = np.random.normal(size=5)

print(x)
```

```
[ 1.62434536 -0.61175641 -0.52817175 -1.07296862  0.86540763]
```

[73]:
```
# Initialize an empty list for storing boolean values
negative = []

# Loop through each element in the array x
for i in range(len(x)):
    if x[i] < 0:
        negative.append(True)
    else:
        negative.append(False)

print(negative)
```

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

[74]:
```
# Shortcut: Vectorized operation to check for negative values
negative = x < 0

print(negative)
```

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

[75]:
```
# Example: Loop through each element in the array x and replace negative values␣
↪with 0
for i in range(len(x)):
    if x[i] < 0:
        x[i] = 0

print(x)
```

```
[1.62434536 0.         0.         0.         0.86540763]
```

[76]:
```
# Shortcut: Replace negative values in x with 0 using a vectorized operation
x[x < 0] = 0

print(x)
```

```
[1.62434536 0.         0.         0.         0.86540763]
```

Extra

Hiding warning message

[77]:
```
import warnings  # importing the warning module

# To hide all warning messages, add the following code line in the cell where␣
 ↪you encountered the warning.
warnings.filterwarnings('ignore')

# To hide a warning based on a category of warning messages, add the additional␣
 ↪category parameter.
warnings.filterwarnings('ignore', category=UserWarning)
```

Categories of Warning

- Deprecation Warning
- User Warning
- Syntax Warning
- Runtime Warning
- ...

Suppressing warning messages can be useful for improving code readability, but it can also hide potentially important information. So, it's recommended to use this approach with caution and only when you are confident that the warnings can be safely ignored.