DSS5201 Data Visualization

Week 2

Yuting Huang

NUS DSDS

2024-08-19

Week 1:

- Python objects
- Python syntax

Week 2:

- Python functions
- Introduction to `numpy`
- Introduction to `pandas`

Recap

The `if` statement executes one or more commands when a specified condition is met.

1. `if`

```
x = 5
if x > 11:
    print("x is greater than 11.")
```

- Note that the print statement will not appear in the console, because the condition is not met.

2. `if... else`

```python
if x > 11:
    print("x is greater than 11.")
else:
    print("x is no larger than 11.")
```

```
x is no larger than 11.
```

3. `if... elif... else`

```
if x > 11:
    print("x is greater than 11.")
elif x > 7:
    print("x is greater than 7, but less than 11.")
else:
    print("x is smaller or equal to 7.")
```

```
x is smaller or equal to 7.
```

For loops iterate over items in a range or any iterable object, such as a list.

```
for i in range(1, 3):
    print(i)
```

```
1
2
```

- After each iteration, the value of `i` will be updated.
- Must use the `print()` function if we want to output a result.

A for loop with if-else statement

```
animals = ["cat", "dog", "dog", "pigeon"]
for i in animals:
    if i == "dog":
        print("This is a dog.")
    else:
        print("These are other animals.")
```

```
These are other animals.
This is a dog.
This is a dog.
These are other animals.
```

A `while` loop is used when we want to perform a task **indefinitely**, until a particular **stopping condition** is met. The flow of execution is:

1. Determine whether the condition is `True` or `False`.
2. If `False`, exist the `while` statement.
3. If `True`, run the statement and go back to step 1.

```
i = 0
while i < 4:
    print(i)
    i += 1
```

```
0
1
2
3
```

Another way to accomplish the same task would be to use `break` to stop the iteration when certain condition is met.

```
i = 0
while True:
    print(i)
    i += 1
    if i >= 4:
        break
```

```
0
1
2
3
```

Both `for` and `while` can be used for iteration, but they serve different purposes and are suited for different scenarios.

- `for` is better when the number of iterations is known in advance.
  - Advantage: Clear and concise; easy to understand.
- `while` is better when the stopping condition is known but the number of iterations is not necessary predetermined.
  - Advantage: More general and flexible.
  - Every `for` loop can be re-written as a `while` loop, but not vice versa.

Note: `while` loops can potentially result in infinite loops if the condition is never met. So we must use them with care.

```python
for i in range(1, 3):
    print(i)
```

```
1
2
```

```python
i = 1
while i < 3:
  print(i)
  i += 1
```

```
1
2
```

Use loops with caution

If loops are not written properly, they can get **very slow** when applied to large data sets or in complex settings.

- As you learn more, you will realize that **vectorization** is preferred over loops since it results in shorter and clearer code.
- We will introduce several `numpy` and `pandas` functions to handle data more efficiently.

Python functions

Before we begin, let's import the following libraries.

```python
import numpy as np
import pandas as pd
```

We have already encountered a few functions in Python.

- As you have noticed, they can take arguments that modify their behavior.
- We can use `help()` to list the arguments of the functions.

```
help(np.median)
```

Key points:

- Arguments without default values must be supplied when calling the function.
- Arguments with default values can be optionally supplied.

The function `np.arange(start, stop, step)` is commonly used to generate regular sequence.

- It generates values from `start` to `stop` (exclusive) with a step size of `step`.
- Note that the stop value is 3.5 to include 3 in the sequence.

```
seq1 = np.arange(1, 3.5, 0.5)
print(seq1)
```

```
[1.  1.5 2.  2.5 3. ]
```

To specify a sequence with a specific number of elements, we use `np.linspace()`.

- The syntax is `np.linspace(start, stop, num)`.
- Note that the ending value is inclusive.

```
seq2 = np.linspace(1, 3.5, 5)
print(seq2)
```

```
[1.    1.625 2.25  2.875 3.5  ]
```

To repeat a single value, we can use list multiplication.

- The following code creates a list containing the value `1` five times.

```
seq3 = [1] * 5
print(seq3)
```

```
[1, 1, 1, 1, 1]
```

To repeat a sequence, we use the `np.tile()` function.

```
seq4 = np.tile(seq3, 2)
print(seq4)
```

```
[1 1 1 1 1 1 1 1 1 1]
```

At some point, we may have to write a **function** of our own.

- A re-usable piece of code that can accept input parameters (arguments).

We will need to decide

1. What argument(s) it should take.
2. Whether these arguments should have default values, and if so, what those default values should be.
3. What output it should return.

The typical approach is to write a sequence of expressions that work, then package them into a function.

Let's define a function called `myfunc1` that takes an input and returns its square.

```
def myfunc1(n):
    n_squared = n ** 2
    return n_squared
```

- In the code, the function name (`myfunc1`) and its argument (`n`) are defined.
    - The argument does not have default value.
- Then we specify what the function should do if it is called. The value (`n_squared`) will be returned after execution.

```
result = myfunc1(2)
print(result)
```

4

In `myfunc1()`, the input argument does not have a default value.

- We will get an error if we don't supply values for it, as Python would not know what value to use.
- If we declare its default value, when we call the function and leave the argument(s) empty, Python will take in the default values.

```
def myfunc2(n = 4):
    n_squared = n ** 2
    return n_squared

result = myfunc2()
print(result)
```

```
16
```

Let us suppose that we wish to write a function to simulate one game of dice between players A and B (from last lecture).

```
np.random.seed(5201)

A = np.random.randint(1, 7)
B = np.random.randint(1, 7)

if A > B:
    print("A")
elif A == B:
    print("Draw.")
else:
    print("B")
```

```
B
```

A function that simulate the game

```python
def single_game():
    A = np.random.randint(1, 7)
    B = np.random.randint(1, 7)

    if A > B:
        return "A"
    elif A == B:
        return "Draw"
    else:
        return "B"
```

Now we can call the function and print the result:

```
# Set seed for reproducibility
np.random.seed(5201)
print(single_game())
```

```
B
```

How about running the single game 1000 times?

```
from collections import Counter
np.random.seed(5201)    # Set seed for reproducibility
results = [0] * 1000    # Initiate a list to store results

for i in range(1000):
    results[i] = single_game()

results_counter = Counter(results) # Summarize results
print(results_counter)
```

```
Counter({'A': 429, 'B': 413, 'Draw': 158})
```

Suppose that each player rolls dice more than one time, and compares the **sum** of his/her dice to the component's.

```
def compare_sum(n_dice = 1):
    A = np.random.randint(1, 7, size = n_dice)
    B = np.random.randint(1, 7, size = n_dice)
    # Compare the sums of the rolls
    if A.sum() > B.sum():
        return "A"
    elif A.sum() == B.sum():
        return "Draw"
    else:
        return "B"
```

```
# Set seed for reproducibility
np.random.seed(5201)

# Compare sum if each player rolls dice three times
results = compare_sum(3)
print(results)
```

```
A
```

Let's practice what we've learned so far.

1. Create a function `website_domain(url)` that grabs the website domain from a URL string.

   For example, if we pass "www.nus.edu.sg" to the function, it should return `nus`.

2. Create a function `divisible(a, b)` that accepts two integers (`a` and `b`). It returns `True` if `a` is divisible by `b` without a remainder.

   For example, `divisible(8, 2)` should return `True` and `divisible(8, 3)` should return `False`.

Coding style

The PEP 8 Python style guide

PEP 8 is the official style guide for Python. It is worth skimming through it.

There are some highlights to remember:

- Use four spaces for indentation.
- Add white spaces around operators. For example, use `x = 1` instead of `x=1`.
- Single and double quotes are both fine for strings.
- Snake case: Variables and functions should be named using `underscores_between_words`.

Other important rules for readability:

- Import all necessary modules at the start of your script.
- Keep your lines of code to a maximum of 79 characters.
- Use comments to explain the code when necessary.
  - Single-line comments start with the # symbol.
  - In-line comments can be placed at the end of a line, also starts with #.
  - Multi-line comments with triple quotes start with `"""` and end with another `"""`.

Introduction to `numpy`

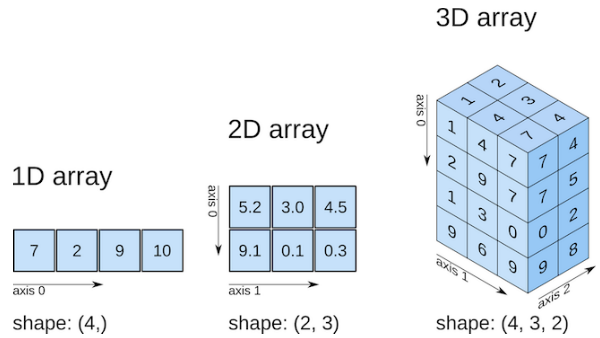
- Introduction to `numpy`.
- `Numpy` arrays.
- Array operations and broadcasting.
- Indexing and slicing.
- Useful `numpy` functions.

Introduction to numpy

`Numpy` stands for **numerical Python**.

- The standard Python library for working with arrays (vectors and matrices), linear algebra, and other numerical computations.
- We usually import pandas with the alias `np`.

```
import numpy as np
```



Arrays are n-dimensional data structures that can contain all the basic Python data types.

- Work best with numeric data.
- Items in an array should be of the same type.

Let's import `numpy` with the alias `np` and create some arrays.

- From existing data (lists), use `np.array()`.
- Or use functions like `np.arange()`, `np.ones()`, `np.zeros()`, …

```
# Create an array from a list
array1 = np.array([1, 2, 3, 4, 5])
print(array1)
```

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

```
# Other ways to create an array
array2 = np.arange(1, 6)
print(array2)
```

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

```python
# A 3x3 array of the number 0
array3 = np.zeros((3, 3))
print(array3)
```

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

```python
# A 2x5 array with random numbers uniformly distributed from 0 to 1
array4 = np.random.rand(2, 5)
print(array4)
```

```
[[0.88347242 0.91388622 0.07810793 0.06848532 0.41396043]
 [0.73495063 0.50540952 0.25923482 0.31867621 0.01407945]]
```

There are three main attributes of an array:

- `.ndim`: The number of dimensions of an array.
- `.shape`: The number of elements in each dimension.
- `.size`: The total number of elements in an array.

```
def print_attributes(x):
    print("Dimensions:", x.ndim)
    print("Shape:", x.shape)
    print("Size:", x.size)
print_attributes(array4)
```

```
Dimensions: 2
Shape: (2, 5)
Size: 10
```

Broadcasting describes how `numpy` treats arrays with different shapes during arithmetic operations.

- Three types of pies at different prices were sold from Friday to Sunday.
- We want to know how much we made per pie type per day.
    - Example: $\$20 \times 2 = \$40$ for apples on Friday.

Pie Cost	
	Cost ($)
Apple	20
Blueberry	15
Pumpkin	25

Number of Pies Sold			
	Friday	Saturday	Sunday
Apple	2	3	1
Blueberry	6	3	3
Pumpkin	5	3	5

Let's create the arrays below:

```
price = np.array([20, 15, 25])
print(price)
```

```
[20 15 25]
```

```
sales = np.array([[2, 3, 1], [6, 3, 3], [5, 3, 5]])
print(sales)
```

```
[[2 3 1]
 [6 3 3]
 [5 3 5]]
```

A mental model

How can we multiply these arrays together?

1. Make the price array and the sales array the same size.
2. Multiply corresponding elements one by one.

Pie Cost			
	Cost ($)	Cost ($)	Cost ($)
Apple	20	20	20
Blueberry	15	15	15
Pumpkin	25	25	25

Number of Pies Sold			
	Friday	Saturday	Sunday
Apple	2	3	1
Blueberry	6	3	3
Pumpkin	5	3	5

```
# Make the price array and the sales array the same size,
price = np.repeat(price, 3).reshape(3, 3)
print(price)
```

```
[[20 20 20]
 [15 15 15]
 [25 25 25]]
```

```
# Element-wise multiplication
revenue = price * sales
print(revenue)
```

```
[[ 40  60  20]
 [ 90  45  45]
 [125  75 125]]
```

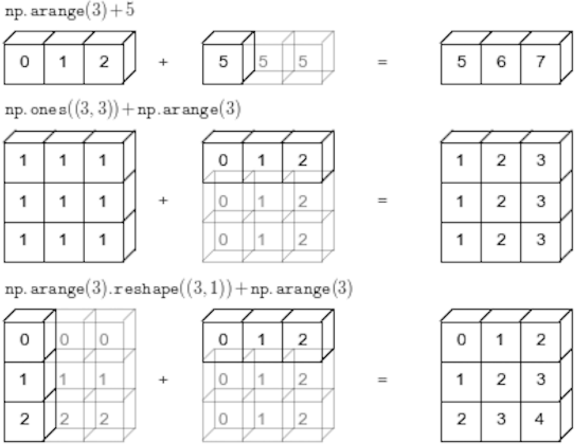
In `numpy`, the smaller array can be **broadcast** across the larger array so they have compatible shapes.

The following code will do the `np.repeat()` internally under the hood.

```
# Reshape the price array
price = np.array([20, 15, 25]).reshape(3, 1)

# Automatic broadcasting for multiplication
revenue = price * sales
print(revenue)
```

```
[[ 40  60  20]
 [ 90  45  45]
 [125  75 125]]
```



Source: Python Data Science Handbook by Jake VanderPlas (2016).

There are three key **reshaping** methods for `numpy` arrays:

- `.reshape()`
- `.ravel()` or `.flatten()`

Let's see some examples.

```
np.random.seed(5201)                        # For reproducibility
x = np.random.randint(9, size = (2, 3)) # A 2 x 3 array
print(x)
```

```
[[3 7 8]
 [1 0 2]]
```

```
# Reshape it into a 3 x 2 array
print(x.reshape(3, 2))
```

```
[[3 7]
 [8 1]
 [0 2]]
```

```
# Flatten an array to a single dimension
print(x.flatten())
```

```
[3 7 8 1 0 2]
```

We've learnt about indexing and slicing for **lists**.

- Doing the same for **arrays** are similar. There are just more dimensions.

```
    Index   Column 0  Column 1  Column 2
Row   0         3         7         8
Row   1         1         0         2
```

```
# Slice the element at index row = 0, column = 2
print(x[0, 2])
```

8

```
# Create a 1d array of 5 random numbers
x = np.random.rand(5)
print(x)
```

```
[0.06848532 0.41396043 0.73495063 0.50540952 0.25923482]
```

```
# Boolean values for elements > 0.5
x_thresh = x > 0.5
print(x_thresh)
```

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

```
# Set all elements > 0.5 to be equal to 0.5
x[x_thresh] = 0.5
print(x)
```

```
[0.06848532 0.41396043 0.5        0.5        0.25923482]
```

This method is useful for handling **missing values** in `numpy` arrays.

```
# Create a 1d array with missing value
x = np.array([1, np.nan, 3, 4])
print(x)
```

```
[ 1. nan  3.  4.]
```

```
# Boolean values for missing values
x_missing = np.isnan(x)

# Replace missing values with 0
x[x_missing] = 0
print(x)
```

```
[1. 0. 3. 4.]
```

Introduction to pandas

Pandas is the most popular Python library for tabular data structures.

- We usually import pandas with the alias pd.

```
import pandas as pd
```

Pandas series

Series is like a `numpy` array but with labels.

- Strictly 1-dimensional and can contain any data type.
- Can be created from a scalar, a list, an array, or a dictionary using `pd.Series()`.

Here are some examples.

Series 1

INDEX	DATA
0	A
1	B
2	C
3	D
4	E
5	F

Series 2

INDEX	DATA
A	1
B	2
C	3
D	4
E	5
F	6

- By default, series are labelled with indices starting from zero.

```
# Default labels
series1 = pd.Series(data = [1, 2, 3, 4, 5, 6])
print(series1)
```

```
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

- We can add a custom index too.

```
# Custom labels
series2 = pd.Series(data = [1, 2, 3, 4, 5, 6],
                    index = ["A", "B", "C", "D", "E", "F"])
print(series2)
```

```
A     1
B     2
C     3
D     4
E     5
F     6
dtype: int64
```

Series are very much like arrays.

- They can be indexed using square brackets `[]` and sliced using the colon `:` notation.

```
# Access the first element
print(series2[0])
```

```
1
```

```
# Access the first three elements
print(series2[0:3])
```

```
A    1
B    2
C    3
dtype: int64
```

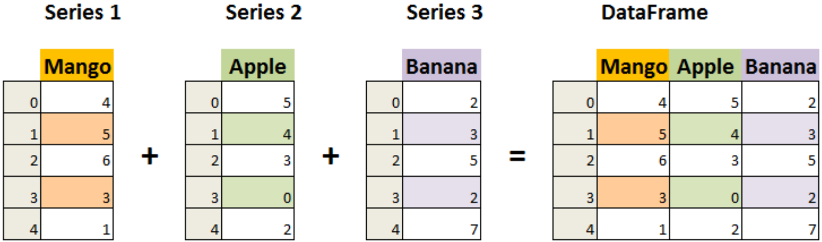
Series 1			Series 2			Series 1 + Series 2	
INDEX	DATA		INDEX	DATA		INDEX	DATA
A	0		-	-		A	NaN
B	1	+	B	10	=	B	11
C	2	+	C	11	=	C	13
D	3	+	D	12	=	D	15
-	-		E	13		E	NaN

Operations between series align values based on their **labels**.

- The resulting index will be the **sorted union** of two indexes.

```
s1 = pd.Series(data = [0, 1, 2, 3], index = ["A", "B", "C", "D"])
s2 = pd.Series(data = [10, 11, 12, 13], index = ["B", "C", "D", "E"])
# Series operation
print(s1 + s2)
```

```
A      NaN
B     11.0
C     13.0
D     15.0
E      NaN
dtype: float64
```



DataFrames are Series stuck together.

- Can be created using `pd.DataFrame()`.
- Index and column labels starts from 0 by default.
- We can use the `index` and `columns` arguments to give them customed labels.

```
# Method 1: From a list
df = pd.DataFrame([["Tom", "Python", 5],
                   ["Mike", "Python", 4],
                   ["Tiffany", "R", 7]],
                   columns = ["Name", "Language", "Courses"])
print(df)
```

```
      Name Language  Courses
0      Tom   Python        5
1     Mike   Python        4
2  Tiffany        R        7
```

```python
# Method 2: From a dictionary
df = pd.DataFrame({"Name": ["Tom", "Mike", "Tiffany"],
                   "Language": ["Python", "Python", "R"],
                   "Courses": [5, 4, 7]})
print(df)
```

```
      Name Language  Courses
0      Tom   Python        5
1     Mike   Python        4
2  Tiffany        R        7
```

There are several ways to select data from a `DataFrame`:

- `[]` and `[[]]`
- `.loc[]` and `.iloc[]`
- Boolean indexing
- `query()`

- A single pair of square brackets selects a single column and returns a Series.

```
# returns a series
df["Name"]
```

```
0        Tom
1       Mike
2    Tiffany
Name: Name, dtype: object
```

Indexing with [[]]

- Double square brackets returns a DataFrame.

```
# returns a DataFrame with one column
df[["Name"]]
```

```
      Name
0      Tom
1     Mike
2  Tiffany
```

```
# returns a DataFrame with two columns
df[["Name", "Language"]]
```

```
      Name Language
0      Tom   Python
1     Mike   Python
2  Tiffany        R
```

`.iloc[]` and `loc[]` are more flexible alternatives for accessing data.

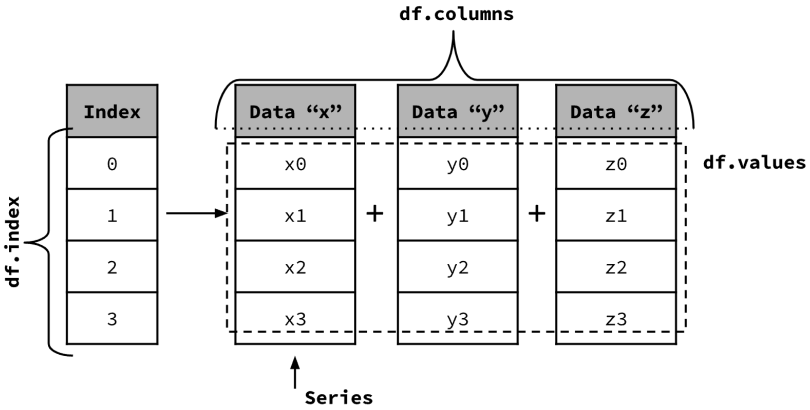
- `.iloc[]` accepts **integer(s)** as references to rows/columns.

```
df.iloc[0]
```

```
Name              Tom
Language       Python
Courses             5
Name: 0, dtype: object
```

```
df.iloc[0:2]
```

```
   Name Language  Courses
0   Tom   Python        5
1  Mike   Python        4
```



df.columns
Index
0
1
2
3
df.index
Data "x"
x0
x1
x2
x3
+
Data "y"
y0
y1
y2
y3
+
Data "z"
z0
z1
z2
z3
df.values
Series

- `.loc[]` accepts **labels** as references to rows/columns.

```
df.loc[:, "Name"]
```

```
0        Tom
1       Mike
2    Tiffany
Name: Name, dtype: object
```

```
df.loc[:, "Name":"Language"]
```

```
      Name Language
0      Tom   Python
1     Mike   Python
2  Tiffany        R
```

Just as with series, we can select data based on **boolean conditions**:

```
df[df["Language"] == "Python"]
```

```
   Name Language  Courses
0   Tom   Python        5
1  Mike   Python        4
```

```
df[(df["Language"] == "Python") & (df["Courses"] > 4)]
```

```
  Name Language  Courses
0  Tom   Python        5
```

`.query()` is a powerful tool for filtering data (similar to SQL).

- It accepts a **string expression** to evaluate the conditions.

```
df.query("Courses > 4 & Language == 'Python'")
```

```
  Name Language  Courses
0  Tom   Python        5
```

- `.query()` also allows us to reference variable in the current workspace.

```
language_choice = "Python"
df.query("Language == @language_choice")
```

```
   Name Language  Courses
0   Tom   Python        5
1  Mike   Python        4
```

- The @ symbol indicates that `language_choice` is defined outside the query string.

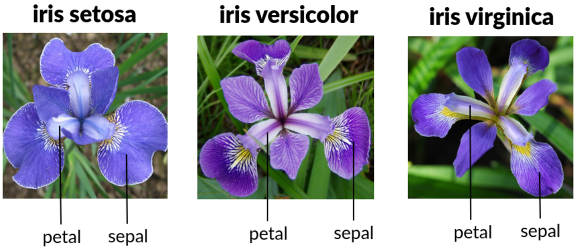
The basics of indexing are as follows:

Operation	Syntax	Output
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[int]`	Series
Slice rows	`df[0:2]`	DataFrame
Select rows by boolean condition	`df[condition]`	DataFrame

Let's examine a classical data set on iris plants.

- Available via the `pydataset` package.
- First we will need to `pip install pydataset`.

```
from pydataset import data
iris = data("iris")
# Define column names
col_names = ["sepal_length_in_cm", "sepal_width_in_cm",
             "petal_length_in_cm", "petal_width_in_cm", "class"]
iris.columns = col_names
```




```
iris.head()
```

```
   sepal_length_in_cm  sepal_width_in_cm  ...  petal_width_in_cm   class
1                 5.1                3.5  ...                0.2  setosa
2                 4.9                3.0  ...                0.2  setosa
3                 4.7                3.2  ...                0.2  setosa
4                 4.6                3.1  ...                0.2  setosa
5                 5.0                3.6  ...                0.2  setosa

[5 rows x 5 columns]
```

There are three helpful attributes/functions for getting **high-level summaires** for `DataFrames`.

- `.shape` gives the shape (rows, cols).
- `.info()` prints information about the `DataFrame`, such as `dtypes`, memory usages, non-null values, etc.
- `.describe()` provides summary statistics of numeric columns (by default).

```
# Shape of DataFrame
iris.shape
```

```
(150, 5)
```

```python
# Info of DataFrame
iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 150 entries, 1 to 150
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   sepal_length_in_cm  150 non-null    float64
 1   sepal_width_in_cm   150 non-null    float64
 2   petal_length_in_cm  150 non-null    float64
 3   petal_width_in_cm   150 non-null    float64
 4   class               150 non-null    object
dtypes: float64(4), object(1)
memory usage: 7.0+ KB
```

```python
# Summary statistics of numeric columns in a DataFrame
iris.describe()
```

```
       sepal_length_in_cm  ...  petal_width_in_cm
count          150.000000  ...         150.000000
mean             5.843333  ...           1.199333
std              0.828066  ...           0.762238
min              4.300000  ...           0.100000
25%              5.100000  ...           0.300000
50%              5.800000  ...           1.300000
75%              6.400000  ...           1.800000
max              7.900000  ...           2.500000

[8 rows x 4 columns]
```

- Here we introduce some common DataFrame operations, e.g., .min(), .mean(), sort_index(), and sort_values().

```
# Min of all numeric columns
iris.min()
```

```
sepal_length_in_cm        4.3
sepal_width_in_cm         2.0
petal_length_in_cm        1.0
petal_width_in_cm         0.1
class                  setosa
dtype: object
```

```
# Min of a selected column
iris["sepal_length_in_cm"].min()
```

```
4.3
```

```python
# Sort values on the index
iris.sort_index(ascending = False).head(10)
```

```
     sepal_length_in_cm  sepal_width_in_cm  ...  petal_width_in_cm      cl
150                 5.9                3.0  ...                1.8  virgin
149                 6.2                3.4  ...                2.3  virgin
148                 6.5                3.0  ...                2.0  virgin
147                 6.3                2.5  ...                1.9  virgin
146                 6.7                3.0  ...                2.3  virgin
145                 6.7                3.3  ...                2.5  virgin
144                 6.8                3.2  ...                2.3  virgin
143                 5.8                2.7  ...                1.9  virgin
142                 6.9                3.1  ...                2.3  virgin
141                 6.7                3.1  ...                2.4  virgin

[10 rows x 5 columns]
```

```python
# Sort values by a specific column
iris.sort_values(by = "sepal_length_in_cm", ascending = True).head(10)
```

```
    sepal_length_in_cm  sepal_width_in_cm  ...  petal_width_in_cm   class
14                 4.3                3.0  ...                0.1  setosa
43                 4.4                3.2  ...                0.2  setosa
39                 4.4                3.0  ...                0.2  setosa
9                  4.4                2.9  ...                0.2  setosa
42                 4.5                2.3  ...                0.3  setosa
23                 4.6                3.6  ...                0.2  setosa
4                  4.6                3.1  ...                0.2  setosa
7                  4.6                3.4  ...                0.3  setosa
48                 4.6                3.2  ...                0.2  setosa
3                  4.7                3.2  ...                0.2  setosa

[10 rows x 5 columns]
```

We can rename selected columns with `.rename()`.

- `inplace = True` modifies the original data frame without creating a new one.

```
# Rename selected columns
iris.rename(
    columns = {"sepal_length_in_cm": "sepal_length",
               "sepal_width_in_cm": "sepal_width"}, inplace = True)
print(iris.columns)
```

```
Index(['sepal_length', 'sepal_width', 'petal_length_in_cm',
       'petal_width_in_cm', 'class'],
      dtype='object')
```

There are two main ways to **add/remove columns**:

- Use [] to add columns.

```
# Add column(s) in the data frame
iris["sepal_length_mm"] = None
iris["sepal_length_mm"] = iris["sepal_length"] * 10
print(iris.head(2))
```

```
   sepal_length  sepal_width  ...   class  sepal_length_mm
1           5.1          3.5  ...  setosa             51.0
2           4.9          3.0  ...  setosa             49.0

[2 rows x 6 columns]
```

- Use `.drop()` to remove columns.

```
# Remove column(s) from the data frame
iris = iris.drop(columns = ["sepal_length_mm"])
print(iris.head(2))
```

```
   sepal_length  sepal_width  petal_length_in_cm  petal_width_in_cm   clas
1           5.1          3.5                 1.4                0.2  setos
2           4.9          3.0                 1.4                0.2  setos
```

Adding/removing rows

We can **add or remove rows** of a data frame in two ways:

- Use `._append()` to add rows.
- Use `.drop()` to remove rows.

Though we won't often be doing this manually, it is nice to have them in our toolbox.

```
# Create another row
another_row = pd.DataFrame([[4.3, 2.0, 1.0, 0.1, "new"]],
                           columns = iris.columns, index = [151])
# Append the row to the bottom of the iris data frame
iris = iris._append(another_row)
print(iris.tail())
```

```
     sepal_length  sepal_width  ...  petal_width_in_cm      class
147           6.3          2.5  ...                1.9  virginica
148           6.5          3.0  ...                2.0  virginica
149           6.2          3.4  ...                2.3  virginica
150           5.9          3.0  ...                1.8  virginica
151           4.3          2.0  ...                0.1        new

[5 rows x 5 columns]
```

```
# Remove the row with a specific index
iris = iris.drop(index = 151)
print(iris.tail())
```

```
     sepal_length  sepal_width  ...  petal_width_in_cm      class
146           6.7          3.0  ...                2.3  virginica
147           6.3          2.5  ...                1.9  virginica
148           6.5          3.0  ...                2.0  virginica
149           6.2          3.4  ...                2.3  virginica
150           5.9          3.0  ...                1.8  virginica

[5 rows x 5 columns]
```

Your Turn: Air quality in NYC

In this exercise, you will examine the daily air quality in New York City in 1973. The data are available via `pydataset`.

```
from pydataset import data
airquality = data("airquality")
```

1. How many rows and columns are their in the data frame? How many months are represented in the data?
2. Create two new data frames, one contains information for summer months (May and June) and one for fall months (July, August, and September). Name them as `df_summer` and `df_fall` respectively. Examine the structure of the new data frames.

Python functions

Introduction to `numpy`:

- Create and inspect an array.
- Reshape an array.

Introduction to `pandas`:

- Create and inspect a data frame.
- Rename columns in a data frame.
- Add/remove columns and rows in a data frame.

Assignment: Pre-course reflections on Canvas by **this Friday 11:59pm**.