

Pandas Series & DataFrames

◆ Reminder

Yesterday we learned **NumPy arrays** → efficient numerical data storage and manipulation.

👉 Pandas **builds on NumPy** to handle **tabular/labeled data** (like Excel tables).

◆ Introduction to Pandas

How Pandas is Related to NumPy

1. Pandas is Built on Top of NumPy

- Under the hood, Pandas **stores data in NumPy arrays**.
 - Every `Series` (1D labeled array (like an Excel column)) in Pandas is basically a **wrapper around a NumPy ndarray** with labels.
 - A `DataFrame` is just a **collection of Series objects**, which means → lots of NumPy arrays working together.
-

2. Shared Operations

- Both Pandas and NumPy support **vectorized operations** (fast, element-wise math without loops).
 - Many Pandas methods actually **call NumPy functions internally**.
Example: `df.mean()` → uses `numpy.mean()`.
-

3. Differences

Feature	NumPy	Pandas
Main Object	<code>ndarray</code> (n-dimensional array)	<code>Series</code> (1D) & <code>DataFrame</code> (2D)
Data Types	Homogeneous (all elements must be same type)	Heterogeneous (different columns can have different types)
Labels	Only integer indices	Row & column labels (names)
Use Case	Numerical computation	Data analysis / tabular data

Use in Data Science

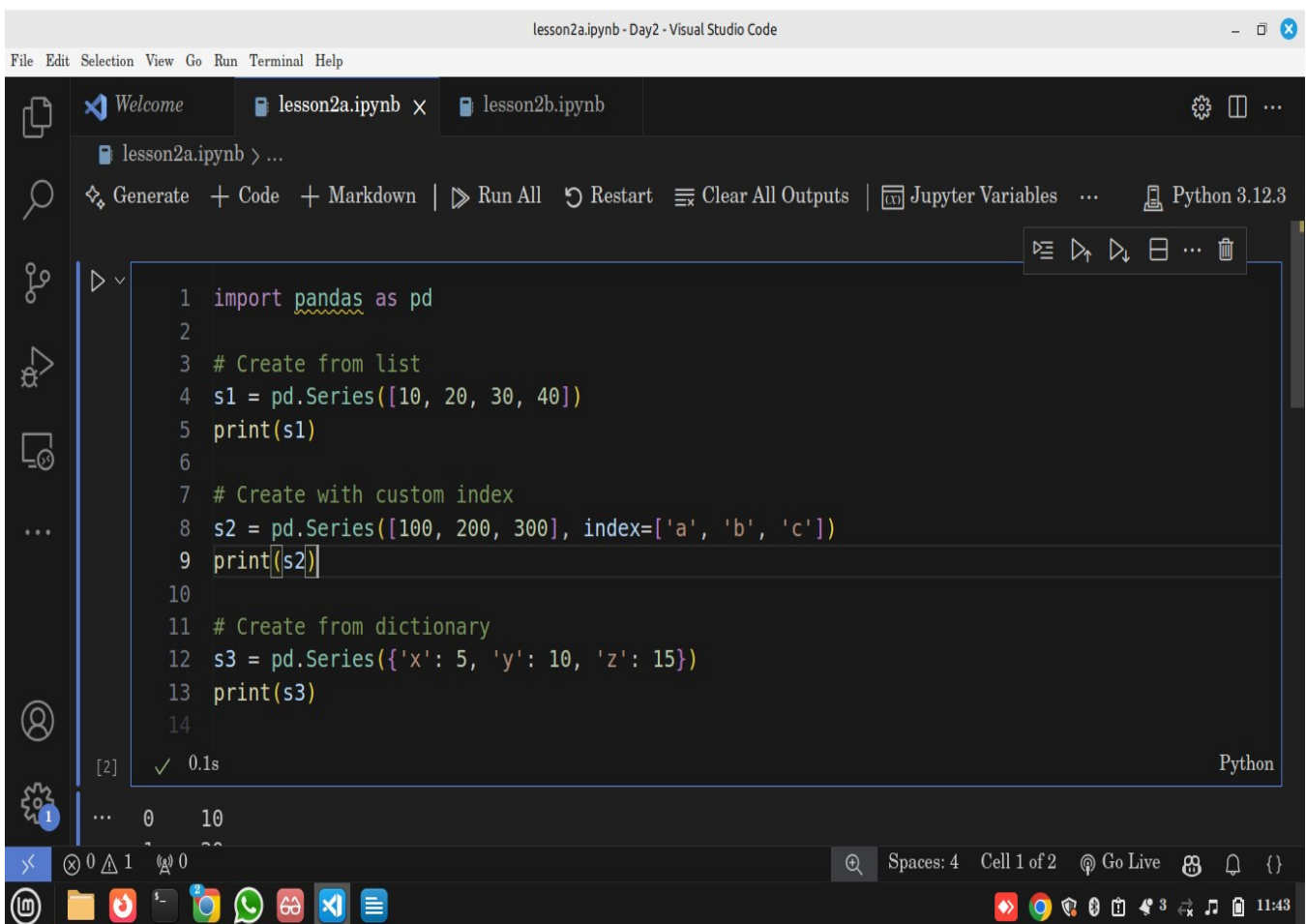
Almost every dataset in data science is loaded into a **DataFrame** before analysis.

◆ Pandas Series

Notes

- Series = **one column** of data, with an **index**.
 - Can be created from lists, NumPy arrays, or dictionaries.
 - Supports slicing, indexing, and operations like NumPy.
-

Practical 1 – Creating a Series

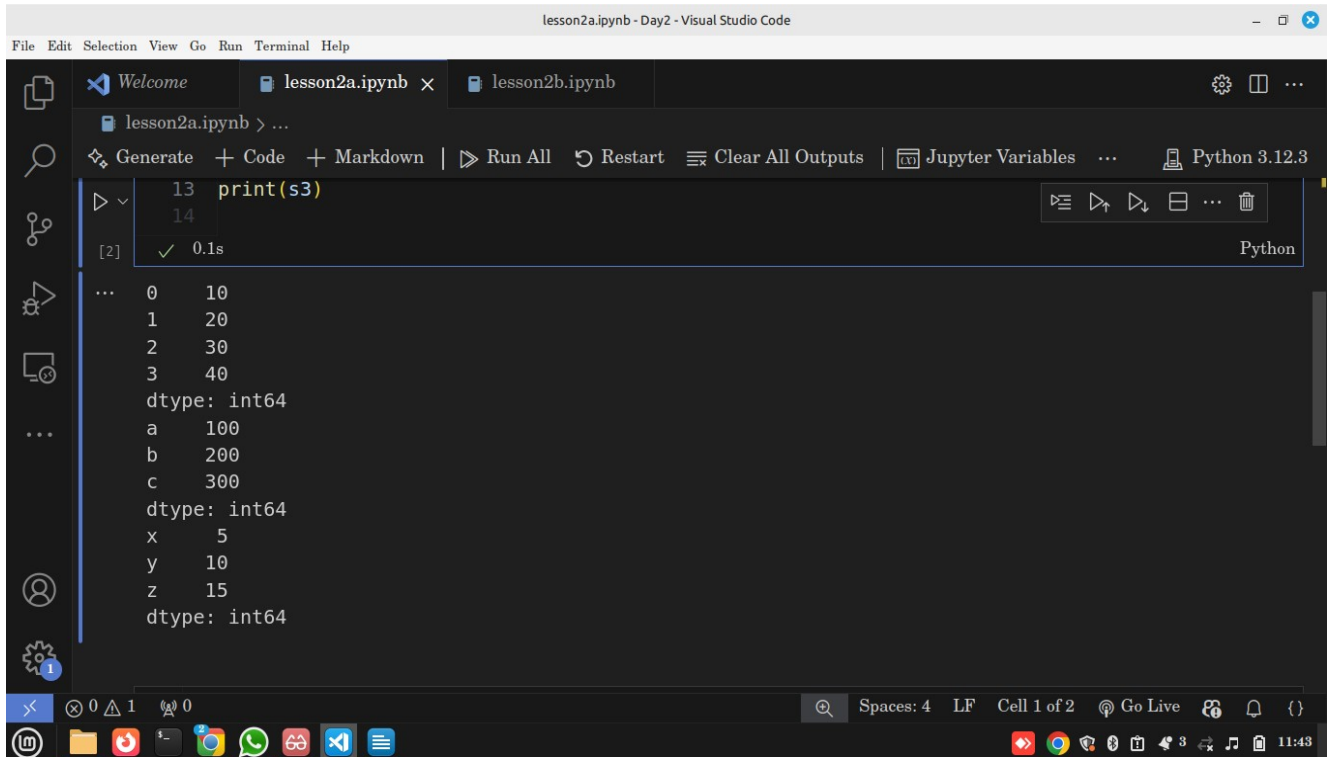


The screenshot shows a Jupyter Notebook titled 'lesson2a.ipynb' in Visual Studio Code. The notebook contains three code cells demonstrating how to create Pandas Series:

```
1 import pandas as pd
2
3 # Create from list
4 s1 = pd.Series([10, 20, 30, 40])
5 print(s1)
6
7 # Create with custom index
8 s2 = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
9 print(s2)
10
11 # Create from dictionary
12 s3 = pd.Series({'x': 5, 'y': 10, 'z': 15})
13 print(s3)
14
```

The output of the first cell is shown as [2] ✓ 0.1s. The bottom status bar indicates 'Spaces: 4', 'Cell 1 of 2', and 'Go Live'.

Output



The screenshot shows a Jupyter Notebook interface in Visual Studio Code. The notebook has two tabs: 'lesson2a.ipynb' and 'lesson2b.ipynb'. The active cell in 'lesson2a.ipynb' contains the code `print(s3)`. The output of this cell is a pandas Series with integer index [0, 1, 2, 3] and values [10, 20, 30, 40]. Below the Series, the data type is shown as `dtype: int64`. The notebook also shows a variable `a` with values [100, 200, 300] and a variable `x` with values [5, 10, 15], both with `dtype: int64`.

```
lesson2a.ipynb > ...  
Generate + Code + Markdown | Run All Restart Clear All Outputs | Jupyter Variables ... Python 3.12.3  
13 print(s3)  
14  
[2] ✓ 0.1s  
...  
0 10  
1 20  
2 30  
3 40  
dtype: int64  
a 100  
b 200  
c 300  
dtype: int64  
x 5  
y 10  
z 15  
dtype: int64
```

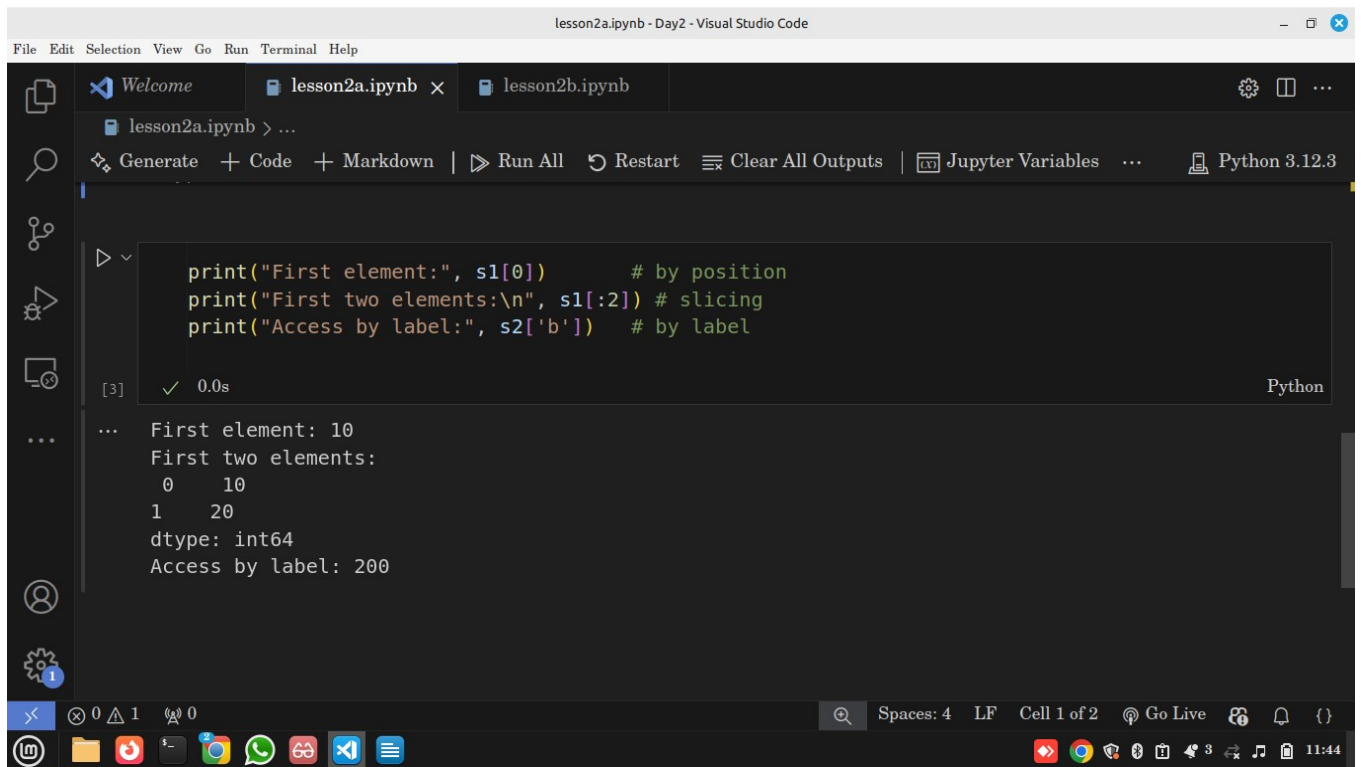
Explanation

- `pd.Series(list)` → creates a Series with default integer index (0,1,2...).
- `index=` lets us name rows (like Excel row labels).
- A dictionary automatically maps **keys** → **index**.

Use in Data Science

Series often represent a **single variable** (like "Age" column in Titanic dataset).

Practical 2 – Indexing and Slicing



The screenshot shows a Jupyter Notebook interface within Visual Studio Code. The notebook is titled 'lesson2a.ipynb' and contains a single code cell. The code cell has three lines of Python code: `print("First element:", s1[0])` with a comment '# by position', `print("First two elements:\n", s1[:2])` with a comment '# slicing', and `print("Access by label:", s2['b'])` with a comment '# by label'. The output of the code cell is displayed below the code, showing the results of the three print statements. The first print statement outputs 'First element: 10'. The second print statement outputs 'First two elements:' followed by a newline and then '0 10' and '1 20'. The third print statement outputs 'Access by label: 200'. The output also shows the data type 'dtype: int64'. The notebook is running on Python 3.12.3. The Visual Studio Code interface includes a sidebar with icons for Explorer, Search, Source Control, and Run and Debug. The bottom status bar shows 'Spaces: 4', 'LF', 'Cell 1 of 2', 'Go Live', and a clock showing '11:44'.

```
print("First element:", s1[0])          # by position
print("First two elements:\n", s1[:2]) # slicing
print("Access by label:", s2['b'])      # by label
```

[3] ✓ 0.0s Python

... First element: 10
First two elements:
0 10
1 20
dtype: int64
Access by label: 200

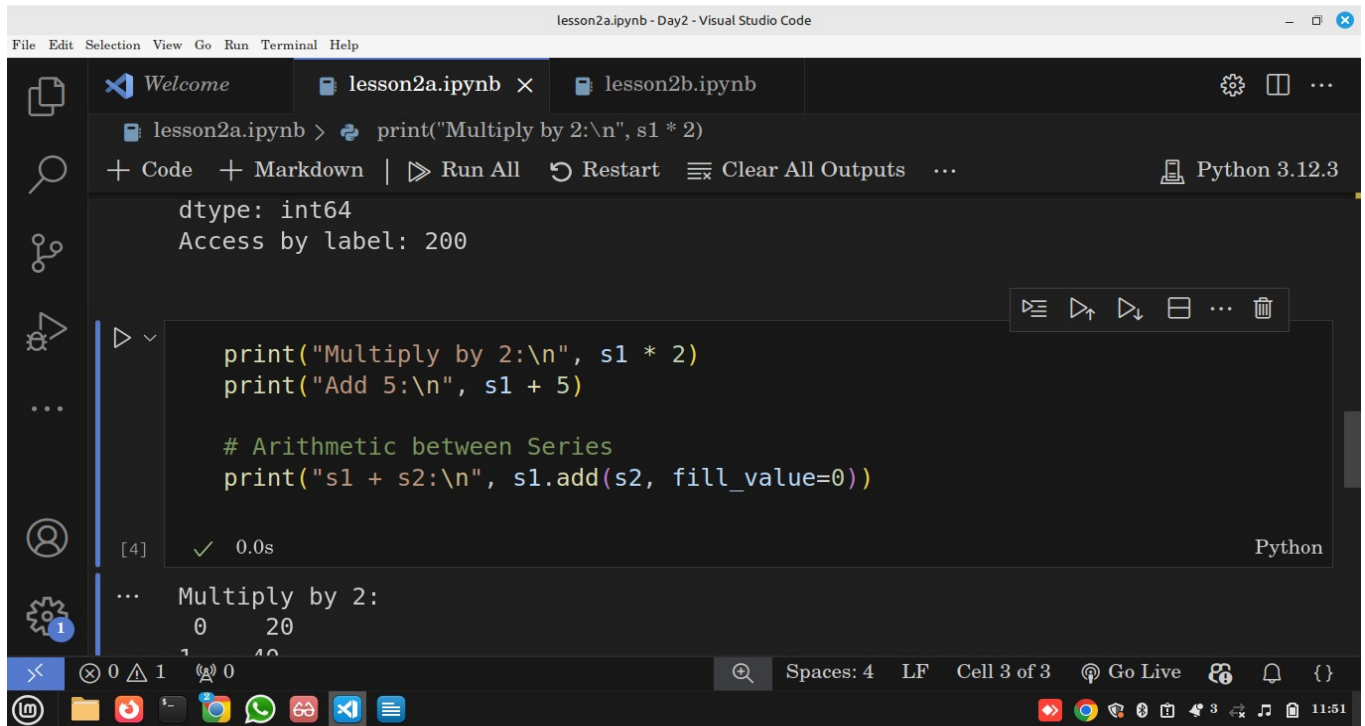
Explanation

- Access by **position** (like NumPy arrays).
- Access by **label** if index is custom.
- Slicing works like Python lists.

Use in Data Science

Lets us extract **specific rows** or **ranges** for analysis.

Practical 3 – Operations



The screenshot shows the Visual Studio Code interface with a Jupyter Notebook open. The notebook has two tabs: 'lesson2a.ipynb' and 'lesson2b.ipynb'. The active tab is 'lesson2a.ipynb', which contains a code cell with the following Python code:

```
print("Multiply by 2:\n", s1 * 2)

dtype: int64
Access by label: 200

print("Multiply by 2:\n", s1 * 2)
print("Add 5:\n", s1 + 5)

# Arithmetic between Series
print("s1 + s2:\n", s1.add(s2, fill_value=0))
```

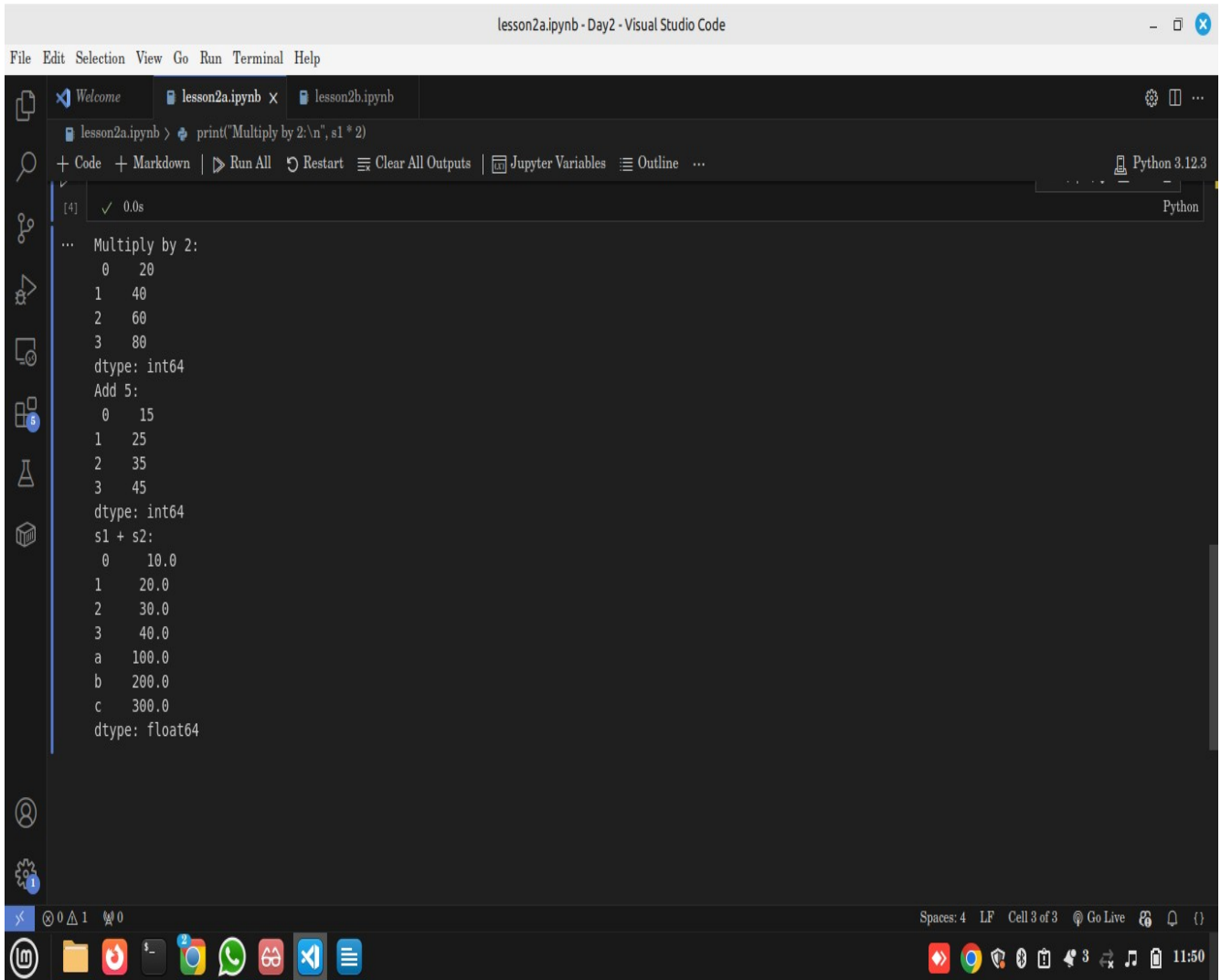
The code cell has been executed, and the output is displayed below it. The output shows the result of the multiplication and addition operations, along with the dtype of the resulting series.

```
[4] ✓ 0.0s Python

... Multiply by 2:
0    20
1    40
```

The bottom status bar indicates the current configuration: Spaces: 4, LF, Cell 3 of 3, Go Live, and a notification icon.

Output



The screenshot shows a Jupyter Notebook interface in Visual Studio Code. The notebook is titled 'lesson2a.ipynb' and contains a single code cell. The code cell has been executed, and the output is displayed below it. The output shows a pandas Series with values [20, 40, 60, 80] and dtype 'int64'. This series is then added to another series with values [15, 25, 35, 45] and dtype 'int64'. The result is a new series with values [35, 65, 95, 125] and dtype 'float64'. The output is formatted as a table with columns 'a', 'b', and 'c'.

```
lesson2a.ipynb > print("Multiply by 2:\n", s1 * 2)

[4] ✓ 0.0s

... Multiply by 2:
0    20
1    40
2    60
3    80
dtype: int64
Add 5:
0    15
1    25
2    35
3    45
dtype: int64
s1 + s2:
0    10.0
1    20.0
2    30.0
3    40.0
a    100.0
b    200.0
c    300.0
dtype: float64
```

Explanation

- Operations apply element-wise.
- When adding Series with different indexes, Pandas aligns by index.
- `fill_value=0` fills missing values before arithmetic.

Use in Data Science

This makes data transformation **vectorized & efficient** compared to Python loops.

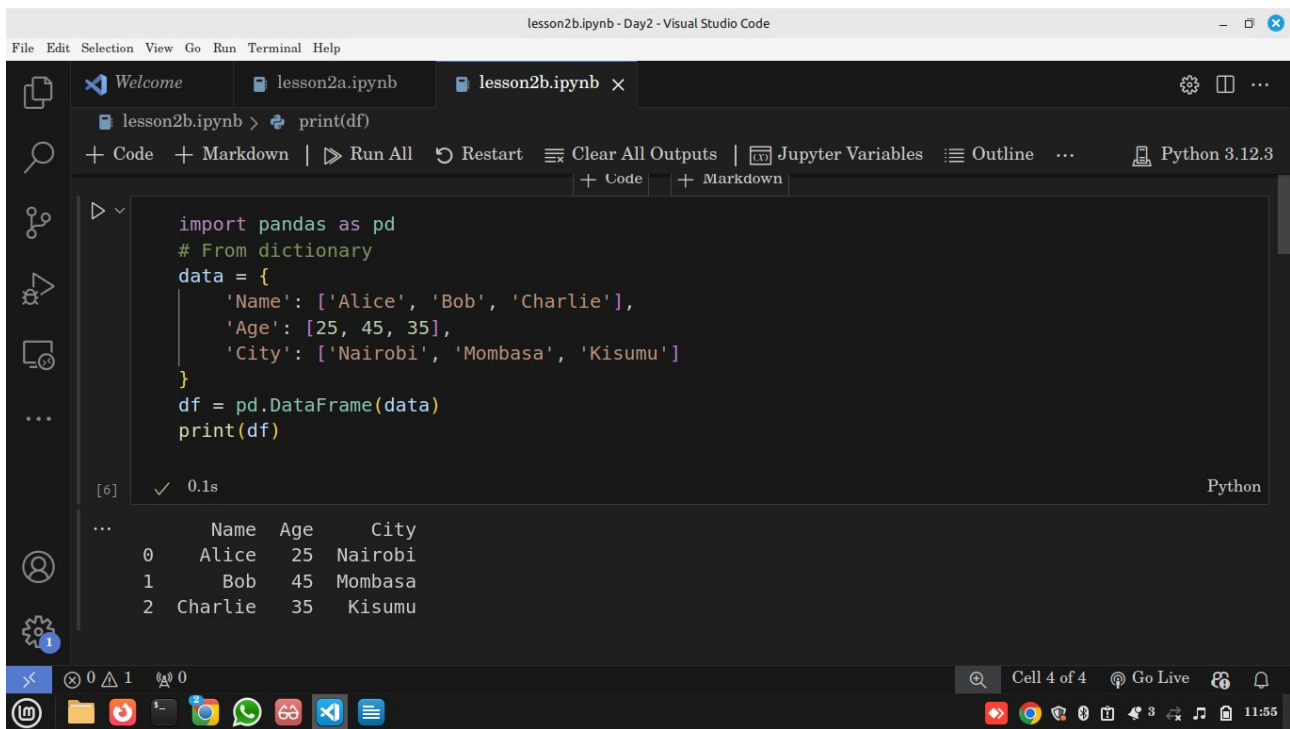
◆ Pandas DataFrames

Notes

- A DataFrame is a **collection of Series** (columns).
- Think of it as a **table**: rows + columns.

- Can be created from dictionaries, NumPy arrays, or files (CSV, Excel, etc.).

Practical 1 – Creating a DataFrame



The screenshot shows a Jupyter Notebook interface within Visual Studio Code. The notebook has two tabs: 'lesson2a.ipynb' and 'lesson2b.ipynb'. The active tab is 'lesson2b.ipynb', which contains a single code cell. The code in the cell is as follows:

```
import pandas as pd
# From dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 45, 35],
    'City': ['Nairobi', 'Mombasa', 'Kisumu']
}
df = pd.DataFrame(data)
print(df)
```

Below the code cell, the output is displayed. It shows a tabular representation of the DataFrame with three columns: 'Name', 'Age', and 'City'. The rows are indexed from 0 to 2.

	Name	Age	City
0	Alice	25	Nairobi
1	Bob	45	Mombasa
2	Charlie	35	Kisumu

The bottom of the screenshot shows the Visual Studio Code interface with various icons and a status bar at the bottom indicating 'Cell 4 of 4', 'Go Live', and the time '11:55'.

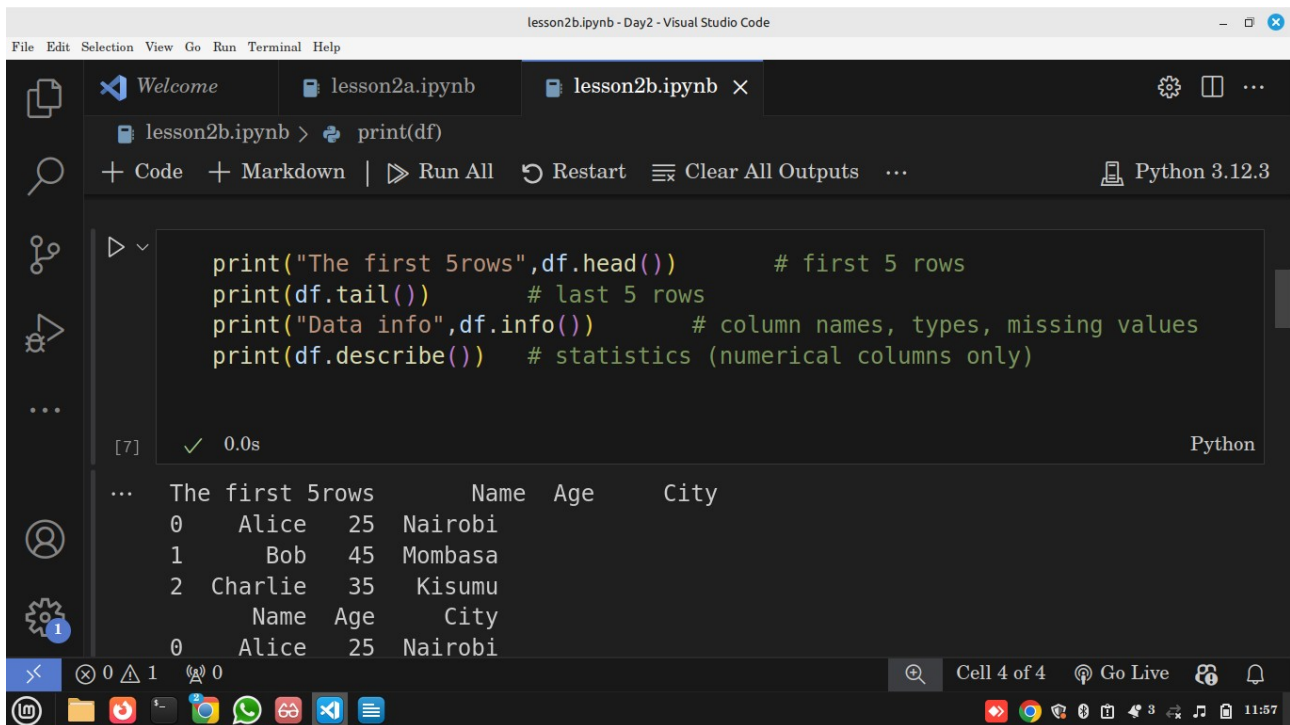
Explanation

- Each key in dict → column name.
- Each list → column values.
- Output is tabular, with row and column labels.

Use in Data Science

DataFrames are the **standard way** to store datasets for analysis.

Practical 2 – Inspecting Data



The screenshot shows a Visual Studio Code window with a Jupyter Notebook. The notebook has two tabs: 'lesson2a.ipynb' and 'lesson2b.ipynb'. The active tab is 'lesson2b.ipynb', which contains the following Python code:

```
print(df)

print("The first 5 rows", df.head())      # first 5 rows
print(df.tail())                          # last 5 rows
print("Data info", df.info())             # column names, types, missing values
print(df.describe())                      # statistics (numerical columns only)
```

The code is executed, and the output is displayed below the code cell. The output shows the first 5 rows of the DataFrame, followed by the last 5 rows, and then the data info and statistics.

```
[7] ✓ 0.0s Python
```

The first 5 rows			Name	Age	City
0	Alice	25	Nairobi		
1	Bob	45	Mombasa		
2	Charlie	35	Kisumu		

Data info			Name	Age	City
0	Alice	25	Nairobi		

The output also shows the statistics (numerical columns only) for the DataFrame.

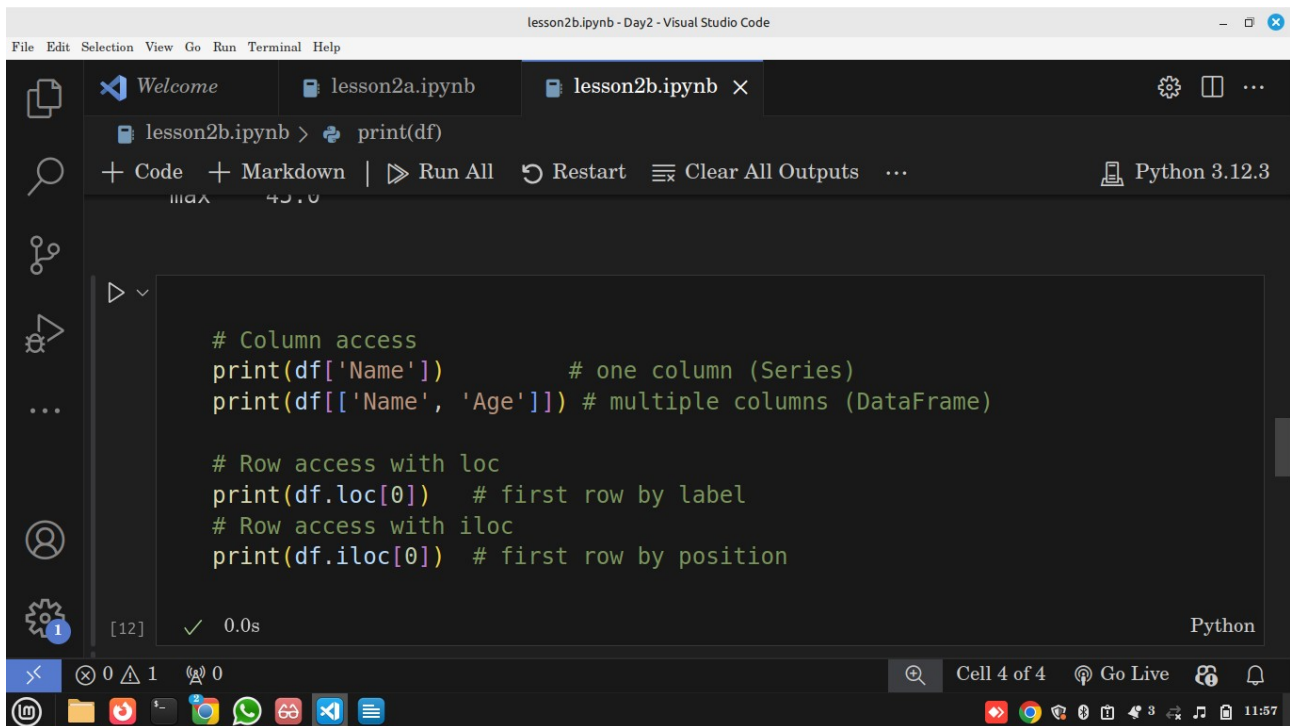
Explanation

- `.head()` and `.tail()` → quick dataset previews.
- `.info()` → useful for checking **data types & null values**.
- `.describe()` → summary statistics.

Use in Data Science

These are the **first commands** analysts run after loading a dataset.

Practical 3 – Accessing Columns & Rows



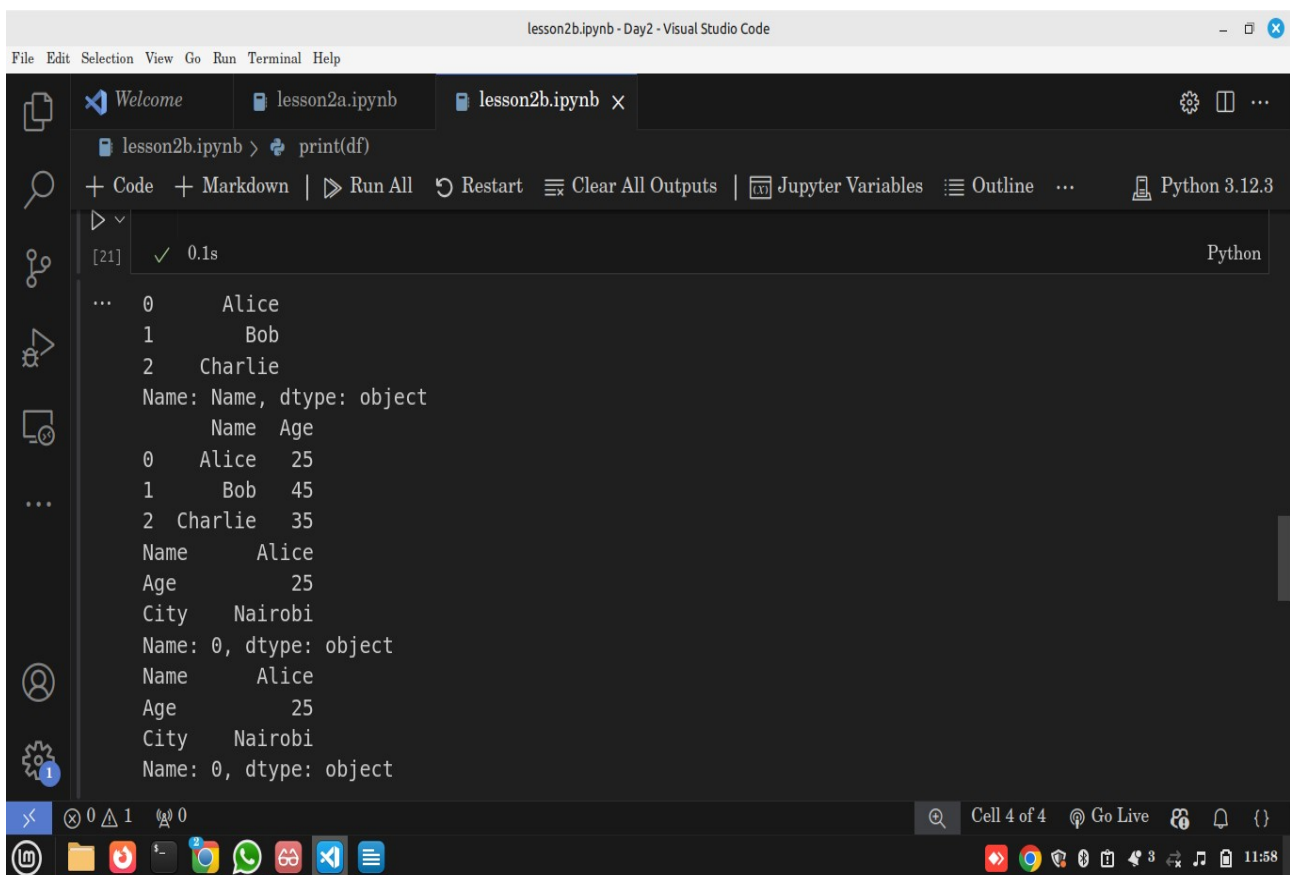
The screenshot shows a Visual Studio Code window with a Jupyter Notebook. The active cell contains the following Python code:

```
# Column access
print(df['Name'])          # one column (Series)
print(df[['Name', 'Age']]) # multiple columns (DataFrame)

# Row access with loc
print(df.loc[0])           # first row by label
# Row access with iloc
print(df.iloc[0])          # first row by position
```

The output of the cell is not visible in this screenshot.

Output



The screenshot shows the same Visual Studio Code window, but now the output of the Jupyter Notebook cell is visible. The output is as follows:

```
[21] ✓ 0.1s

... 0      Alice
    1      Bob
    2      Charlie
Name: Name, dtype: object
     Name Age
0      Alice 25
1      Bob  45
2      Charlie 35
Name      Alice
Age        25
City      Nairobi
Name: 0, dtype: object
Name      Alice
Age        25
City      Nairobi
Name: 0, dtype: object
```

Explanation

- Single column = Series.
- Multiple columns = DataFrame.
- `.loc` → label-based indexing.
- `.iloc` → position-based indexing.

Use in Data Science

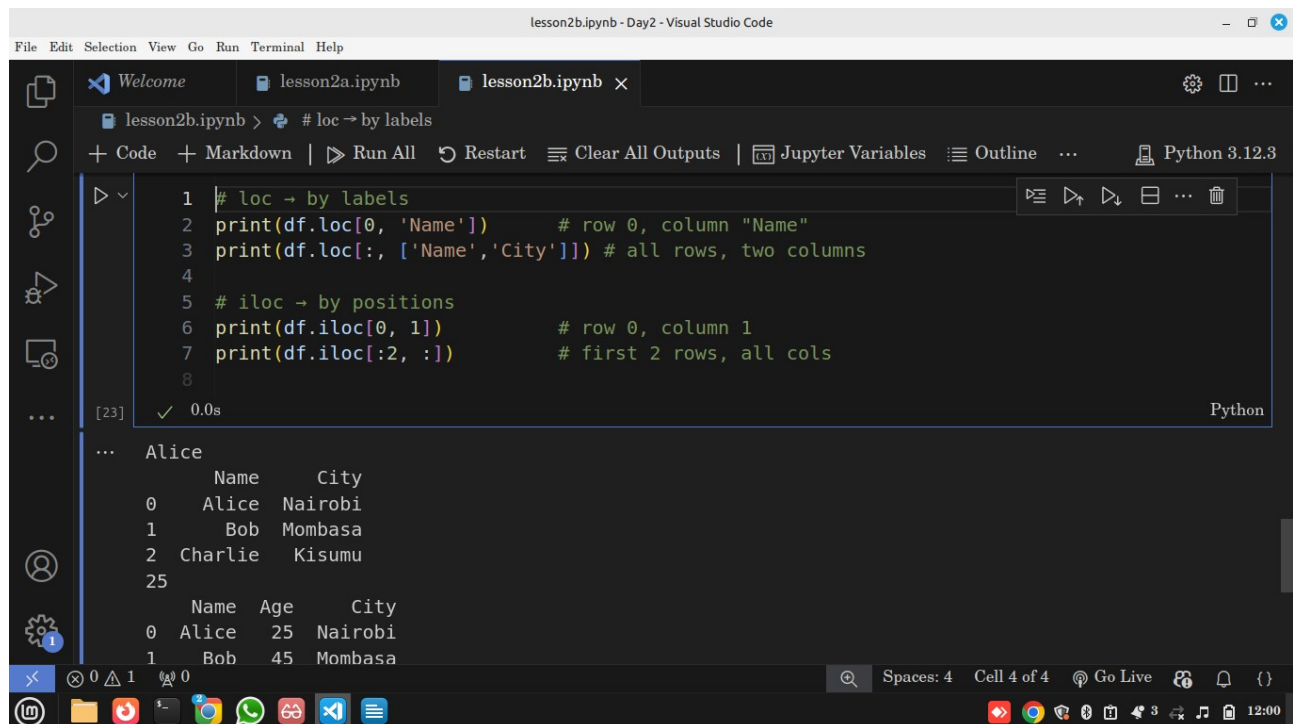
Lets us **select subsets** for analysis (e.g., only numeric columns for statistics).

◆ Indexing: loc vs iloc

Notes

- `.loc[]` → label-based (row/column names).
 - `.iloc[]` → integer-based (row/column positions).
-

Practical



The screenshot shows a Jupyter Notebook interface within Visual Studio Code. The notebook is titled 'lesson2b.ipynb' and contains a single code cell. The code demonstrates pandas indexing using both `.loc` and `.iloc`. The output of the code cell shows the results of these operations on a DataFrame.

```
1 # loc → by labels
2 print(df.loc[0, 'Name'])      # row 0, column "Name"
3 print(df.loc[:, ['Name', 'City']]) # all rows, two columns
4
5 # iloc → by positions
6 print(df.iloc[0, 1])          # row 0, column 1
7 print(df.iloc[:2, :])         # first 2 rows, all cols
8
```

The output of the code cell is as follows:

```
[23] ✓ 0.0s

... Alice
      Name  City
0  Alice  Nairobi
1    Bob  Mombasa
2  Charlie  Kisumu
25
      Name  Age  City
0  Alice   25  Nairobi
1    Bob   45  Mombasa
```

Explanation

- `.loc[0, 'Name']` → fetches "Alice".
- `.iloc[0, 1]` → fetches first row, second column (Age = 25).

Use in Data Science

Used heavily in **subsetting datasets** for training/testing ML models.

Reflection

Why are DataFrames more powerful than NumPy arrays?