Introduction to loc and iloc

intro

Purpose: Access and modify data in pandas DataFrames.

Key Difference:

loc: Select data by labels (index or column names).

iloc: Select data by integer positions (like Python lists).

Why is it Important:

Efficient data manipulation.

Clear syntax for row and column selection.

Using loc

Key Features:

- Select rows/columns by their **labels**.
- Supports slicing, conditions, and multiple labels.

```
# Access a row by index label
df.loc["row1"]

# Access a column by name
df.loc[:, "column_name"]

# Access a subset of rows and columns
df.loc["row1":"row3", ["column1", "column2"]]
```

➢ Intuitive for labeled data.

Using iloc

Key Features:

- Select rows/columns by their integer positions.
- Works like slicing in Python.

```
# Access a row by position
df.iloc[0]

# Access a column by position
df.iloc[:, 2]

# Access a range of rows and columns
df.iloc[1:4, 0:2]
```

➢ Useful for numeric-only indexing.

Comparing loc and iloc

Feature	loc	iloc
Selection	By labels (names)	By integer positions
Syntax	df.loc[row_label, col_label]	df.iloc[row_pos, col_pos]
Flexibility	Works with labels, slices, and conditions	Works with numeric slices only

```
df.loc["row2", "col3"]  # By label
df.iloc[1, 2]            # By position
```

Common Use Cases

loc:

Filter rows based on a condition:

```
df.loc[df["age"] > 30]
```

Access specific columns:

```
df.loc[:, ["name", "age"]]
```

iloc:

Select rows and columns by position:

```
df.iloc[1:5, 0:3]
```

Useful for numeric indexing in loops.

Tips and Best Practices

- Prefer **loc** for labeled data, especially with string-based indices or conditions.
- Use **iloc** for positional indexing, especially when indices are numeric or unnamed.
- **Hybrid Approach**: Combine both when necessary:

```
# Select columns by position but use `loc` syntax
df.loc[:, df.columns[1:3]]
```