awesome—let's make you dangerous with `pandas` indexing. below is a practical, example-rich guide to **`.loc`** (label-based) and **`.iloc`** (position-based): how they work, every common scenario, and gotchas.

---

setup: a tiny dataframe we'll use in examples

```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    {
        "city": ["London", "London", "Leeds", "Leeds", "Bristol"],
        "year": [2023, 2024, 2023, 2024, 2024],
        "sales": [120, 140, 90, 95, 110],
        "cost": [70, 80, 55, 60, 65],
    },
    index=["a","b","c","d","e"]  # explicit row labels
)
# df:
#   city     year  sales  cost
# a London   2023    120    70
# b London   2024    140    80
# c Leeds    2023     90    55
# d Leeds    2024     95    60
# e Bristol  2024    110    65
```

---

mental model

- **`.loc[row_selector, col_selector]`** → **labels** (index/column names).
    - Slices are **inclusive** (end label included).
    - Boolean masks align **by index labels**.
- **`.iloc[row_selector, col_selector]`** → **integer positions** (0..n-1).
    - Slices are **exclusive** at the end (Python convention).
    - Boolean masks are **positional** (length must match).

You can pass:

- a single label/position
- a list/array of labels/positions
- a slice
- a boolean mask
- a **callable** returning any of the above

`.loc` — label-based indexing (the workhorse)

1) select rows by label

```
df.loc["b"]               # row with index label "b"
df.loc[["b","d"]]         # multiple labels
```

2) select rows and columns by label

```
df.loc["b", "sales"]                 # scalar (row "b", column "sales") → 140
df.loc[["b","d"], ["city","sales"]]# 2 rows × 2 cols
```

3) slice by label (end is included)

```
df.loc["b":"d", "city":"sales"]
# rows b, c, d and columns city, year, sales (inclusive of "sales")
```

4) boolean row filter (mask aligns by index)

```
mask = df["city"].str.startswith("L")  # a,b,c,d → True for London/Leeds
rows
df.loc[mask, ["city","sales"]]
```

5) boolean + multiple conditions

```
df.loc[(df["city"]=="London") & (df["year"]==2024), :]
```

6) callable indexers (nice for readable pipelines)

```
df.loc[lambda x: x["sales"] > x["cost"] * 1.5, ["city","sales","cost"]]
```

7) set/assign with `.loc` (safe, avoids chained indexing)

```
# create a margin column
df.loc[:, "margin"] = df["sales"] - df["cost"]

# conditional assignment
df.loc[df["sales"] >= 120, "tier"] = "A"
df.loc[df["sales"] < 120, "tier"] = "B"
```

8) add a new row by label (if your index isn’t unique, be careful)

```
df.loc["f"] =
{"city":"Cardiff","year":2024,"sales":105,"cost":66,"margin":39,"tier":"B"}
```

9) select columns only (row “:”)

```
df.loc[:, ["city","year"]]      # all rows, selected columns
df.loc[:, "sales":"margin"]     # label slice across columns (inclusive end)
```

10) reindexing-style safe selection (missing labels raise `KeyError`)

```
df.loc[["a","x"], :]   # KeyError because "x" doesn’t exist
# Use .reindex for “allow missing” behaviour:
df.reindex(index=["a","x"], columns=["city","sales"])
```

11) `.loc` with DatetimeIndex (bonus)

When index is dates, label slices are calendar-aware & inclusive:

```
ts = df.set_index(pd.to_datetime(["2024-01-01","2024-02-01","2024-02-
10","2024-03-05","2024-03-20"]))
# ts.loc["2024-02"]   # all rows in Feb 2024
```

12) MultiIndex selection

```
# example multiindex
m = df.set_index(["city","year"]).sort_index()
# single key tuple
m.loc[("Leeds", 2023), :]
# partial key with slice
m.loc[("Leeds", slice(None)), :]
# using IndexSlice for complex selections
idx = pd.IndexSlice
m.loc[idx[["Leeds","London"], 2024], ["sales","cost"]]
```

`.iloc` — position-based indexing (zero-based integers)

1) basic row/column by position

```
df.iloc[1]            # 2nd row (index "b")
df.iloc[1, 2]         # row 2, col 3 (sales) → 140
df.iloc[[0,2,4], [0,2]]  # pick specific positions
```

2) slice by position (end excluded)

```
df.iloc[1:4, 1:3]     # rows 1..3 and cols 1..2
```

3) steps and negative indices

```
df.iloc[::-1, :]      # reverse rows
df.iloc[:, ::-1]      # reverse columns
df.iloc[-2:, -2:]     # last 2 rows × last 2 cols
```

4) boolean mask by position (length must match)

```
pos_mask = np.array([True, False, True, False, True])
df.iloc[pos_mask, :]   # keeps rows 0,2,4
```

5) callable with `.iloc` (returns positions)

```
df.iloc[lambda x: [0, -1], :]   # first and last row by position
```

6) assignment by position

```
df.iloc[:, 2] = df.iloc[:, 2] * 1.1   # increase sales by 10%
```

`.loc` vs `.iloc` — quick contrasts

Topic	`.loc`	`.iloc`
Selection basis	**Labels** (index/column names)	**Integer positions**
Slice end	**Inclusive**	**Exclusive**
Boolean mask	**Aligns by index labels**	**Positional**, same length
Supports callables	Yes	Yes
Missing labels	`KeyError`	N/A (positions always exist if in range)
Typical use	data wrangling, conditions, named columns	algorithmic/positional slicing, negative index tricks

---

power patterns & idioms

compute new columns with conditions

```
df.loc[:, "gp"] = np.where(df["sales"] >= 120, "high", "low")
```

filter-then-select columns

```
(df.loc[df["city"].eq("London"), ["year","sales"]]
   .sort_values("year"))
```

select rows by list membership

```
df.loc[df["city"].isin(["Leeds","Bristol"]), :]
```

regex / substring filters

```
df.loc[df["city"].str.contains(r"^L", case=False, na=False), :]
```

safe single-cell get/set (faster scalars): `.at` / `.iat`

```
df.at["b", "sales"]   # like loc but scalar
df.iat[1, 2]          # like iloc but scalar
df.at["b","sales"] = 150
```

avoid “SettingWithCopyWarning”

Don’t chain like `df[df["sales"]>100]["tier"]="A"` (may edit a view).
Do:

```
mask = df["sales"] > 100
df.loc[mask, "tier"] = "A"
```

selecting columns by dtype then using `.loc`

```
num_cols = df.select_dtypes(include="number").columns
df.loc[:, num_cols].mean()
```

keep top-k per group (with `.loc` and sort_values`)

```
top2 = (df.sort_values(["city","sales"], ascending=[True, False])
          .groupby("city")
          .head(2))
```