MONTGOMERY COLLEGE

Mathematics, Statistics, and Data Science

Data 201

Intro to Data Analysis in Python

Python Fundamentals for R Users

Week 1

Dr. Rebin Muhammad

Welcome — Week 1

You already think in data and functions. This week we map that mindset to Python: what stays the same, what changes, and how to avoid the classic traps.

Today (75 min): philosophy, objects & types, indexing, NumPy, and a short R→Python translation.

Have the **Week 1 Notebook** open so we can try things as we go.

What we'll do today (Week 1)

1. **Philosophy** — Why Python feels “explicit” (and why that’s good)
2. **Objects & types** — How Python represents data (and missing values)
3. **The big gotcha** — Indexing: 0-based, and why it trips up R users
4. **NumPy** — Your R-like vectorized world in Python
5. **Practice** — Translate a short R script into Python

Everything connects to the Week 1 Notebook.

Learning goals

By the end of today you will be able to:

1. **Explain** the main philosophical differences between R and Python
2. **Create and manipulate** basic Python objects (including types and missing values)
3. **Use NumPy** for vectorized computation
4. **Translate** simple R code into Python and run it

R vs Python: one table

	R	Python
Built for	Statistics	General-purpose
How things work	Many defaults (`mean`, `lm`, …)	You import and name the tool
Mental model	“It just works”	“I choose the library”

Bottom line: Explicitness is a feature. It scales better and avoids surprises.

“Explicit” in one example

R: The language gives you `mean`.

```r
mean(x)
```

Python: You say where it comes from.

```python
import numpy as np
np.mean(x)
```

- No magic: every function has a clear home (e.g. `np.*`, `pd.*`).
- In big projects that reduces “where did this come from?” confusion.

➡ Notebook Section 1

Everything has a type

Python makes types visible: `int`, `float`, `bool`, `str`, `None`.

Why it helps: Fewer silent coercions. When something breaks, the error usually points at a type mismatch.

```python
type(5)       # <class 'int'>
type(5.0)     # <class 'float'>
type("hello") # <class 'str'>
type(None)    # <class 'NoneType'>
```

➡ Notebook Section 2

Missing values: two kinds in Python

R	Python
`NA` (one concept)	`None` + `np.nan` (two roles)

- **`None`** — “no value” in general. Not for numeric math.
- **`np.nan`** — “missing number.” Propagates in arithmetic; used in NumPy and pandas.

```
import numpy as np
np.nan + 1    # nan
None + 1      # TypeError
```

For data work: use **`np.nan`** in numeric arrays and **`pd.isna()`** in pandas.

➡ Cheat Sheet, Section 7

Core data structures (the short version)

Python	R analogue	Use in this course
List `[1, 2, 3]`	List / vector	General container; *not* for vectorized math
Tuple `(1, 2)`	Vector (conceptually)	Immutable sequences
Dict `{"a": 1}`	Named list	Key–value storage
NumPy array	Vector	**This is where we do math**
pandas DataFrame	`data.frame`	Next class

Takeaway: For numeric operations, use **NumPy arrays**, not plain lists.

➡ Notebook Section 3

The gotcha: indexing is 0-based

	R	Python
First element	`x[1]`	`x[0]`
Slicing	`x[1:3]` → indices 1 and 2	`x[1:3]` → indices 1 and 2 (3 is excluded)

Python: Start at 0; slice `a:b` means “from `a` up to but not including `b`.”

```
x = [10, 20, 30, 40]
x[0]     # 10  (first)
x[-1]    # 40  (last)
x[1:3]   # [20, 30]
```

Most common R→Python bug: off-by-one. When in doubt, check the first and last index.

➡ Notebook Section 4

Control flow: indentation is syntax

No braces. **Colon + indent** define the block.

```python
if x > 0:
    print("positive")
else:
    print("zero or negative")
```

- Use 4 spaces (or stay consistent). The language enforces structure.
- Same indent = same block; deeper indent = nested block.

➡ Notebook Section 4

Loops vs vectorization

In R you often rely on vectorized operations. **In Python:**

- **Loops** are explicit: `for`, `while`.
- **Vectorization** comes from **NumPy**: use arrays, not lists.

```python
# Not vectorized (list)
[1, 2, 3] * 2    # [1, 2, 3, 1, 2, 3]  — repetition!

# Vectorized (NumPy)
import numpy as np
np.array([1, 2, 3]) * 2    # array([2, 4, 6])
```

Rule: Numeric work → `np.array(...)` then operate.

➡ Notebook Section 5

Why NumPy is the foundation

- **Speed** — Implemented in C; element-wise and array operations are fast.
- **Broadcasting** — Clear rules for combining shapes (no silent recycling).
- **Ecosystem** — pandas and scikit-learn are built on NumPy.

```python
import numpy as np
x = np.array([1, 2, 3])
x + 10          # array([11, 12, 13])
x * [1, 2, 3]   # array([1, 4, 9])
```

➡ Notebook Section 5

Broadcasting vs recycling

R	Python (NumPy)
Recycles the shorter vector to match length	**Broadcasting**: strict rules, often no recycling
Can hide length mismatches	Shape mismatches → errors (safer)

Trade-off: Python can feel stricter. In return you get fewer silent wrong results.

When you see a shape/broadcast error: check lengths and dimensions, or expand explicitly (e.g. `reshape`).

Pitfall 1: “Why doesn’t `mean(x)` work?”

R: `mean(x)` is built in.

Python: There is no built-in `mean` for arrays. Use the library:

```python
import numpy as np
x = np.array([1, 2, 3])
np.mean(x)    # 2.0
```

Habit: Start data-analysis scripts with `import numpy as np` (and `import pandas as pd` when you need tables).

Pitfalls 2–4 (quick list)

- **Lists and math** — `[1,2,3] + 4` is not “add 4 to each”; use `np.array([1,2,3]) + 4`.
- **Forgetting imports** — NumPy and pandas are not loaded by default. Import at the top.
- **Off-by-one** — First element is `x[0]`. When a result looks wrong, check indices.

➡ More in Notebook Section 6

Active learning (15–20 min): R → Python

R code:

```
x <- c(2, 4, 6, 8)
y <- ifelse(x > 4, x/2, x*2)
mean(y)
```

Your tasks:

1. **Predict** — What is `y`? What is `mean(y)`? (Do it in your head or on paper.)
2. **Translate** — Write the Python version (NumPy: `np.where`, `np.mean`).
3. **Run** — Execute in the notebook and confirm.

➡ Class 1 Notebook, Section 7

Active learning – solution

R: `ifelse(condition, yes, no)`

Python: `np.where(condition, yes_array, no_array)`

```python
import numpy as np
x = np.array([2, 4, 6, 8])
y = np.where(x > 4, x / 2, x * 2)    # [4, 8, 3, 4]
np.mean(y)    # 4.75
```

Compare with your answer and with a neighbor. Any questions on `np.where` or indexing?

Debrief: did we hit the goals?

1. **Explain** — Python is explicit (imports, types, libraries); NumPy gives you vectorized computation.
2. **Create & manipulate** — Types, `None` vs `np.nan`, lists vs arrays, 0-based indexing.
3. **Use NumPy** — `np.array`, `np.mean`, `np.where`, and “use arrays for math.”
4. **Translate** — You turned R’s `c()`, `ifelse()`, and `mean()` into Python.

Questions?

Next week: Data wrangling with pandas

- **dplyr** → **pandas** — `filter`, `select`, `mutate`, `group_by`, `summarize`
- **Pipes** → **method chaining** — `.query()`, `.assign()`, `.groupby()`, `.agg()`
- **Bring** your R → Python cheat sheet

Week 1's objects and NumPy are the foundation everything else builds on.

Thank you