Python Lecture 1–2: Introduction to Python

Basic Objects and NumPy Arrays | Computing in Statistics

Dr. Yen-Yi Ho

Department of Statistics, University of South Carolina

Outline

1. Why Python? Getting oriented
2. Basic Python objects: lists, tuples, dictionaries
3. NumPy arrays: creating, indexing, slicing
4. Array attributes, reshaping, and combining arrays
5. Arithmetic and summary statistics on arrays
6. Practice

Why Python?

- A general-purpose programming language that has become one of the dominant languages for data science, alongside R
- **Dynamically typed**: you don't have to declare the type of a variable before using it
- Huge ecosystem of libraries: **NumPy** (arrays), **pandas** (data frames), **matplotlib** (plots), **scikit-learn**, **PyTorch**, ...
- We will lean on the analogy with R throughout this course — most ideas transfer, but the syntax (and a few defaults) differ

Importing Libraries

Unlike base R, Python's core language is intentionally small. To do most anything useful, you **import** a library.

- `numpy` is the name of the library
- `np` is the **alias** we give it — by convention, (almost) everyone uses `np`
- After this, every NumPy function is accessed as `np.something`

Tip

Tip: Use `?function` (in IPython/Jupyter) to pull up help for a function, just as with `?function` in R.

Outline

1. Why Python? Getting oriented
2. **Basic Python objects: lists, tuples, dictionaries**
3. NumPy arrays: creating, indexing, slicing
4. Array attributes, reshaping, and combining arrays
5. Arithmetic and summary statistics on arrays
6. Practice

Three basic container types

Python has three fundamental container objects:

- **list** — ordered, mutable, made with `[ ]`
- **tuple** — ordered, immutable, made with `( )`
- **dictionary** — key–value pairs, made with `{ }`

We will then move on to **NumPy arrays**, which behave more like R's vectors and matrices.

The Python List

The **list** is the basic Python "container" object — built with square brackets:

> ⚠ Warning
>
> **Watch out!** Python indexing starts at **0**, not 1 as in R!

```
'b'
```

Lists: Printing, Concatenating, Coercing

```
The first value in y is 'a'.
```

```
'1'
```

A list can build a sequence with `range`, and can mix types freely:

```
str
```

Lists: Multiple Assignment and Editing

Important contrast with R: the `*` operator on a list does *not* do entrywise arithmetic — it repeats the list!

```
[1, 2, 3, 1, 2, 3]
```

This is one of several signs that lists are not built for numerical computing — that's what NumPy arrays are for.

The Python Tuple

A **tuple** is like a list, but **immutable** — it cannot be edited after creation. Built with parentheses:

```
['hey', 'hi', 'ahoy', 'sup']
```

- Trying to edit a tuple's entry raises an error
- Immutability makes tuples more memory-efficient
- We will mostly see tuples used as *arguments* to functions (e.g. specifying the shape of an array)

The Python Dictionary

A **dictionary** stores **key–value** pairs, accessed by key rather than by position:

```
'2:20 - 3:35 pm'
```

Values can be edited by key:

For most of our statistics and data-science work, we will rely much more on **NumPy arrays** than on lists or dictionaries.

Outline

1. Why Python? Getting oriented
2. Basic Python objects: lists, tuples, dictionaries
3. **NumPy arrays: creating, indexing, slicing**
4. Array attributes, reshaping, and combining arrays
5. Arithmetic and summary statistics on arrays
6. Practice

Creating NumPy Arrays

NumPy arrays are more like R's vectors and matrices: entries must all be the **same type**.

`numpy.ndarray`

Building sequences, similarly to `seq()` in R:

Mixed types get **upcast** (e.g. integers become floats) so the array stays a single type.

Creating Arrays from Scratch

```
array([[0, 9, 2],
       [6, 9, 0],
       [0, 8, 8]])
```

- Give `(rows, columns)` as a **tuple** for shape
- A matrix (2-d array) is made from a list of equal-length lists: `np.array([[1,2,3],[4,5,6]])`

Random Number Generators

The currently recommended way to draw random numbers is via a **generator** object:

This mirrors `rpois()`, `runif()`, `rbinom()` in R, but the distribution's parameters are accessed as *methods* of the generator object.

Accessing Array Entries

```
-1.0
[-0.5  -0.25]
[-0.25  0.    0.25  0.5   0.75]
[-1.   -0.75 -0.5  -0.25]
[-1.   -0.25  0.5 ]
0.75
```

> **Note**
>
> **Key difference from R:** In Python, a negative index counts entries from the *end*; it does *not* drop an entry as it does in R.

Indexing 2-D Arrays (Matrices)

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

De-selecting rows/columns uses ~ rather than a negative index:

```
[4 5 7]
```

```
array([1, 3, 5, 7])
```

Outline

1. Why Python? Getting oriented
2. Basic Python objects: lists, tuples, dictionaries
3. NumPy arrays: creating, indexing, slicing
4. **Array attributes, reshaping, and combining arrays**
5. Arithmetic and summary statistics on arrays
6. Practice

Array Attributes

Attributes describe an array's dimensions and storage type; access with a `.` (no parentheses):

```
dtype('int64')
```

Every NumPy array has a single **dtype**: common ones are `int64`, `float64`, `bool_`.

> **Warning**
>
> **Watch out!** Assigning a float into an integer array silently truncates the value — no warning is given!

Reshaping Arrays

```
array([[0.  , 0.05, 0.1 ],
       [0.15, 0.2 , 0.25],
       [0.3 , 0.35, 0.4 ],
       [0.45, 0.5 , 0.55],
       [0.6 , 0.65, 0.7 ],
       [0.75, 0.8 , 0.85],
       [0.9 , 0.95, 1.  ]])
```

- `reshape` fills the new array **across rows** first
- Use `np.transpose(A)` (or `A.transpose()`) to fill across columns instead
- Arrays can have more than 2 dimensions, just as in R

```
array([[0.  , 0.35, 0.7 ],
       [0.05, 0.4 , 0.75],
       [0.1 , 0.45, 0.8 ],
       [0.15, 0.5 , 0.85],
       [0.2 , 0.55, 0.9 ],
       [0.25, 0.6 , 0.95],
```

Methods: Applying Functions to Objects

Many NumPy functions can also be called as a **method** attached to the object with a `.`:

```
10
```

This "`object.method()`" pattern has no direct analogue in base R and shows up constantly in Python.

Aliases vs. Copies

A **subset** of an array is an **alias**, not a new array — editing it edits the original!

To get an independent copy, append `.copy()`:

Subsetting with a boolean mask

```
array([[ 0,  0,  0,  0,  0,  5],
       [ 6,  0,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Combining and Splitting Arrays

```
array([  1,   2,   3,  98,  99, 100])
```

```
array([[0.         , 0.         , 0.         , 0.68101127, 0.8
        0.85747239],
       [1.         , 1.         , 0.         , 0.16889923, 0.2
        0.49270278]])
```

Splitting is the reverse operation:

Outline

1. Why Python? Getting oriented
2. Basic Python objects: lists, tuples, dictionaries
3. NumPy arrays: creating, indexing, slicing
4. Array attributes, reshaping, and combining arrays
5. **Arithmetic and summary statistics on arrays**
6. Practice

Arithmetic on NumPy Arrays

The operators `+ - * /` and `**` (exponent), `%` (modulus) work **entrywise**, as in R:

```
array([1, 2, 2])
```

> ⚠ Warning
>
> **No recycling!** Unlike R, NumPy will *not* silently recycle a shorter array to match a longer one — mismatched shapes raise an error.

Common math functions live in NumPy: `np.exp`, `np.log`, `np.sin`, `np.arctan`, ...

Summary Statistics on Arrays

```
array([11.68044301, 13.40424463,  8.868334  ])
```

- `axis=0` operates **down** columns; `axis=1` operates **across** rows
- `np.sum` is much faster than Python's built-in `sum` on large arrays

Missing Values

Create a missing value with `None`; most summary functions have a `nan`-aware version:

```
array([0.00849215, 0.1892947 , 0.01073326])
```

Compare: `np.sum` → `np.nansum`, `np.mean` → `np.nanmean`, etc. — the same pattern as `na.rm = TRUE` in R, just with a different function name instead of an argument.

Outline

1. Why Python? Getting oriented
2. Basic Python objects: lists, tuples, dictionaries
3. NumPy arrays: creating, indexing, slicing
4. Array attributes, reshaping, and combining arrays
5. Arithmetic and summary statistics on arrays
6. **Practice**

Practice: Write Code

1. Write code to create the list `[0, 3, 6, 9, 12, 0, 3, 6, 9, 12]`.
2. Create the NumPy array whose ith row (starting at $i = 0$) is filled with the value i, repeated 4 times, for $i = 0, \ldots, 7$.
3. Write code to build the $(n-1) \times n$ "successive difference" matrix with -1 on the main diagonal and 1 on the diagonal just above it, and zeros elsewhere, for any n.
4. Simulate 10,000 rolls of a pair of six-sided dice with `rng.integers(low=1, high=7, size=(10000,2))`. Find the proportion of rolls whose sum is odd.

Practice: Read Code

Predict the output of each code chunk before running it.

```
'cow'
```