#1 Numpy, Pandas Basics

AI Training

Numpy

Okay, let's pause for a sec!

Python checklist:

At the moment, **python** is ruling the fields of data science and machine learning, after a long period of domination by R.

So you definitely need to know this language to be able to use machine learning efficiently.

Checklist:

- Variables and objects
- Conditions
- Loops
- Basic input and output
- Functions
- Python data structures
  - Lists
  - Tuples
  - Dictionary

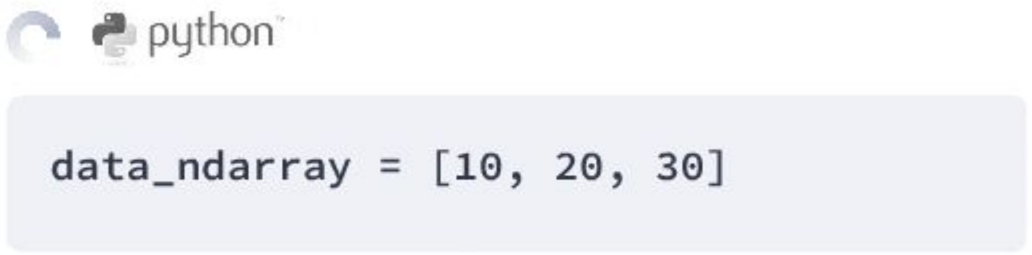
Python Refresher Course:

Intro to Python

Back to Numpy!

Introduction

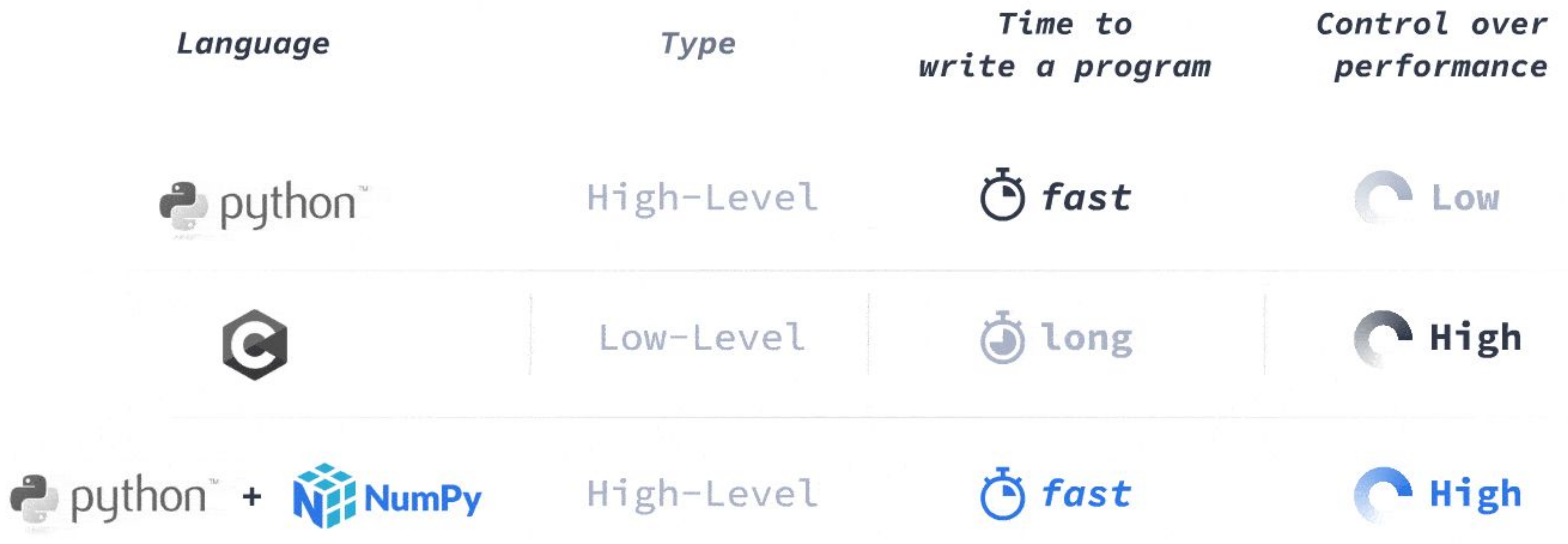
When it comes to programming in Python, libraries are where the real power is. Similar to plugins or extensions, libraries unlock your code and make it more efficient. With a few fundamentals of Python programming, you can use libraries for high-performance data analysis and data visualization. In this lesson, we'll go over one of the most user-friendly libraries out there: NumPy.



```
data_ndarray = [10, 20, 30]
```

Introduction (Cont'd)

NumPy is very popular because it makes writing programs easy. Python is a high-level language, which means you don't have to allocate memory manually. With low-level languages, you have to define memory allocation and processing, which gives you more control over performance, but it also slows down your programming. NumPy gives you the best of both worlds: processing performance without all the allocation.



Language	*Type*	*Time to write a program*	*Control over performance*
python	High-Level	*fast*	Low
C	Low-Level	long	High
python + NumPy	High-Level	*fast*	High

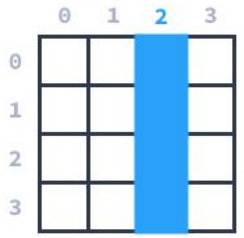
Takeaways

You'll need to be comfortable with programming in Python, but here are a few takeaways you can expect in this lesson:

- How vectorized operations speed up your code
- How to select data from NumPy ndarrays
- How to analyze data using NumPy methods

Selecting Columns

Selecting a Single Column | **List of lists method** | **NumPy method**

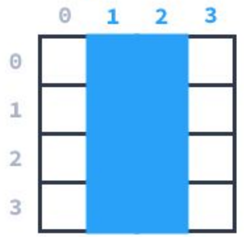


```
sel_lol = []

for row in data_lol:
    col3 = row[2]
    sel_lol.append(col3)
```

```
sel_np = data_np[:,2]
```

- Produces a 1D ndarray.

Selecting Multiple Columns

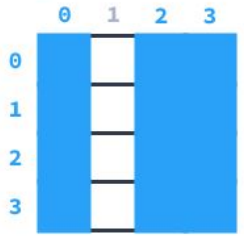


```
sel_lol = []

for row in data_lol:
    col23 = row[1:3]
    sel_lol.append(col23)
```

```
sel_np = data_np[:,1:3]
```

- Produces a 2D ndarray.

Selecting Multiple, Specific Column



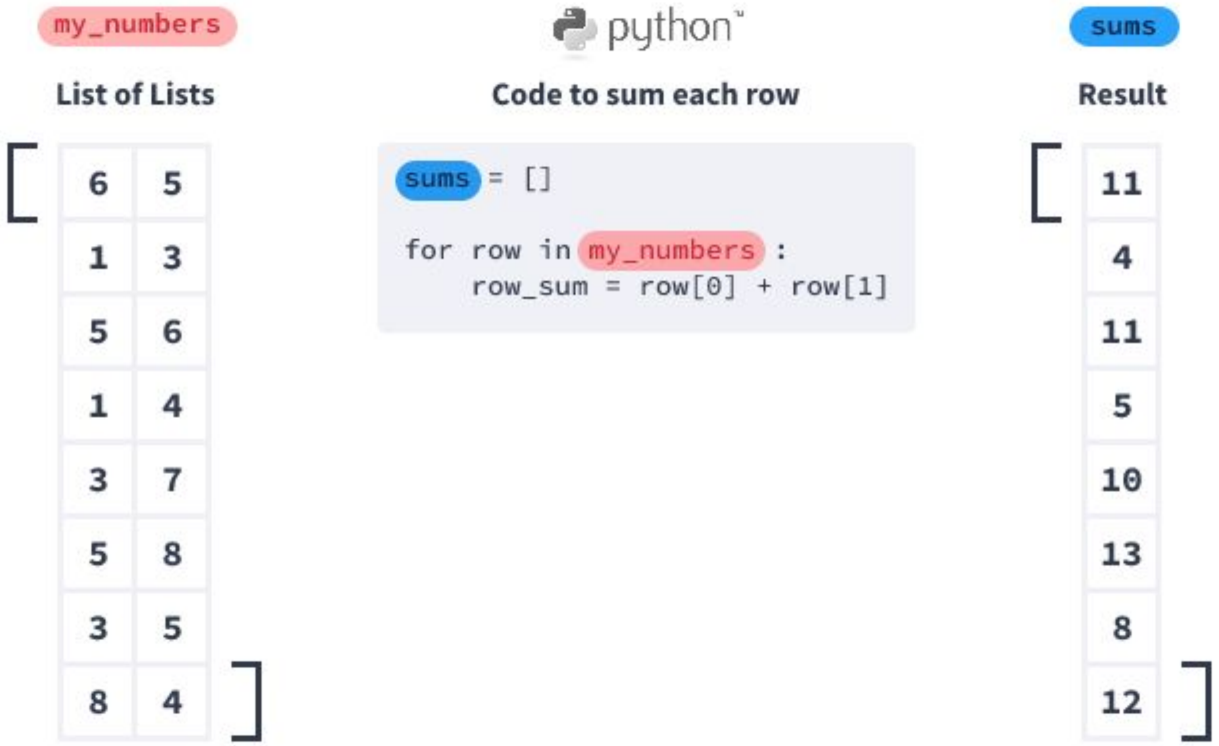
```
sel_lol = []

for row in data_lol:
    cols = [row[0],
            row[2],
            row[3]]
    sel_lol.append(cols)
```

```
cols = [0,2,3]
sel_np = data_np[:,cols]
```

- Produces a 2D ndarray.

Vector Math (Python)

With data in a list of lists, we'd have to construct a for-loop and add each pair of values from each row individually:



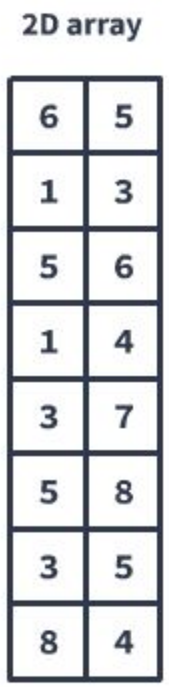
6	5
1	3
5	6
1	4
3	7
5	8
3	5
8	4

```
sums = []

for row in my_numbers :
    row_sum = row[0] + row[1]
```

11
4
11
5
10
13
8
12

Vectorization (Numpy)

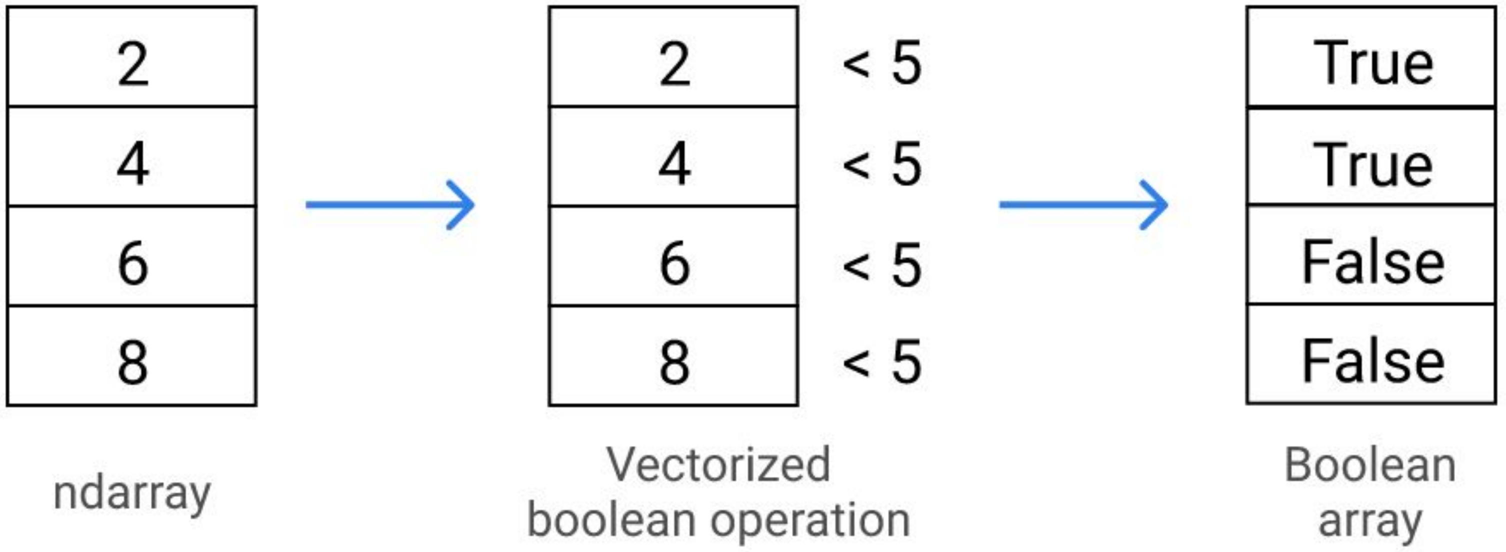
Here's what would happen behind the scenes (with **Numpy**):



Boolean Arrays

```
>>> print(np.array([2,4,6,8]) < 5)  => [True True False False]
```

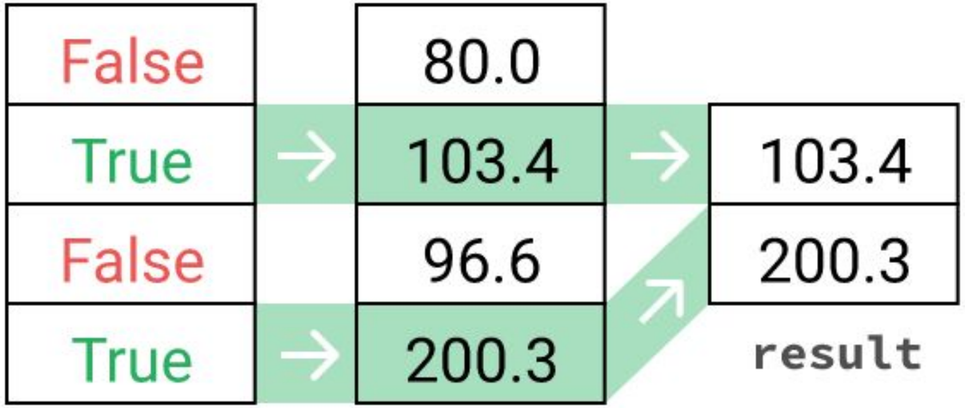
Each value in the array is compared to five. If the value is less than five, True is returned. Otherwise, False is returned.



Boolean Indexing

```
>>> c = np.array([80.0, 103.4,
                  96.9, 200.3])

>>> c_bool = c > 100  => [False True False True]
```

```
result = c[c_bool]
```

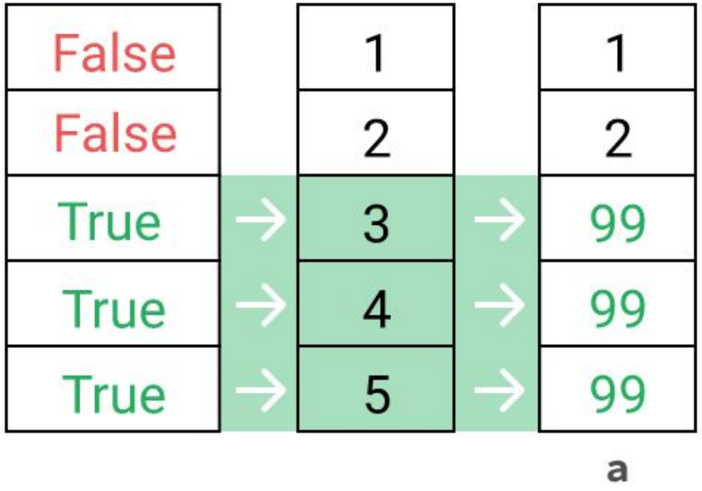


Assignment Using Boolean Arrays

```
>>> a = np.array([1, 2, 3, 4, 5])
```

Notice in the diagram below that we used a "shortcut" — we inserted the definition of the Boolean array directly into the selection. This "shortcut" is the conventional way to write Boolean indexing.

```
a[a > 2] = 99
```



Broadcasting

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations **so that looping occurs in C instead of Python**.

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([2.,  4.,  6.])

>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2.,  4.,  6.])
```

General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1.

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes.

Input arrays do not need to have the same *number* of dimensions. The resulting array will have the same number of dimensions as the input array with the greatest number of dimensions, where the size of each dimension is the largest size of the corresponding dimension among the input arrays. Note that missing dimensions are assumed to have size one.

Practice

Scientific Computing in Python

Arrays, not loops – the NumPy way of thinking.

Intro to Numpy

Pandas

Introduction

Although NumPy provides fundamental structures and tools that make working with data easier, there are limitations for its usefulness:

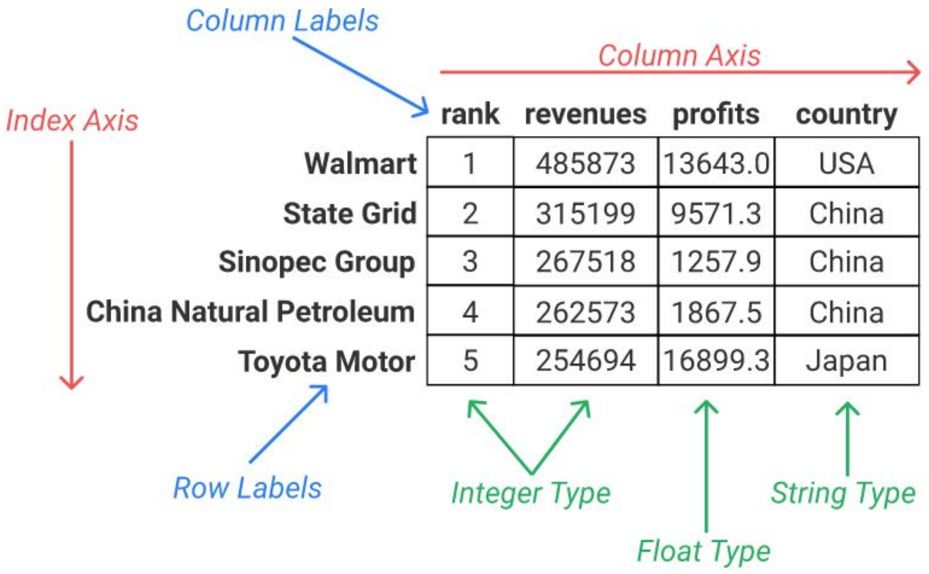
- The lack of support for column names forces us to frame questions as multi-dimensional array operations.
- Support for only one data type per ndarray makes it more difficult to work with data that contains both numeric and string data.
- There are many low level methods, but there are many common analysis patterns that don't have pre-built methods.

The pandas library provides solutions to all of these pain points and more. **Pandas is not a replacement for NumPy, but extension of NumPy**. The underlying code for pandas uses the NumPy library extensively, which means the concepts you've been learning will come in handy as you begin to learn more about pandas.

Introduction to Dataframe

The primary data structure in pandas is called a dataframe. Dataframes are the pandas equivalent of a Numpy 2D ndarray, with a few key differences:

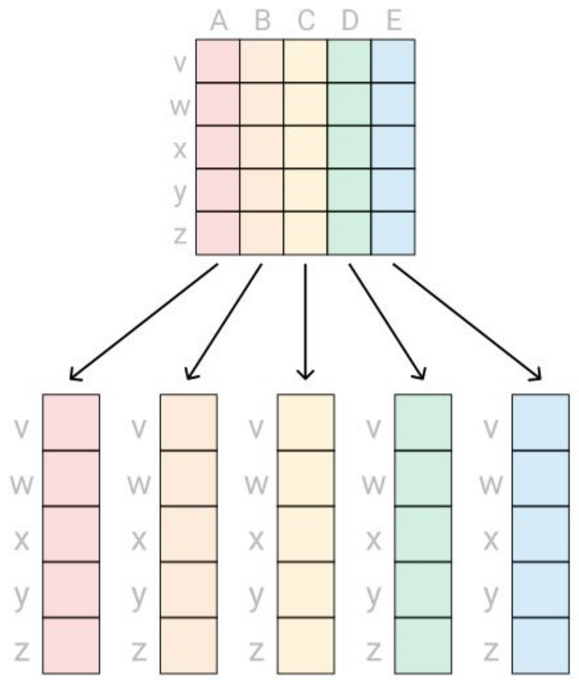
- Axis values can have string labels, not just numeric ones.
- Dataframes can contain columns with multiple data types: including integer, float, and string.



	rank	revenues	profits	country
Walmart	1	485873	13643.0	USA
State Grid	2	315199	9571.3	China
Sinopec Group	3	267518	1257.9	China
China Natural Petroleum	4	262573	1867.5	China
Toyota Motor	5	254694	16899.3	Japan

Introduction to Series

Series is the pandas type for one-dimensional objects. Anytime you see a 1D pandas object, it will be a series. Anytime you see a 2D pandas object, it will be a dataframe.



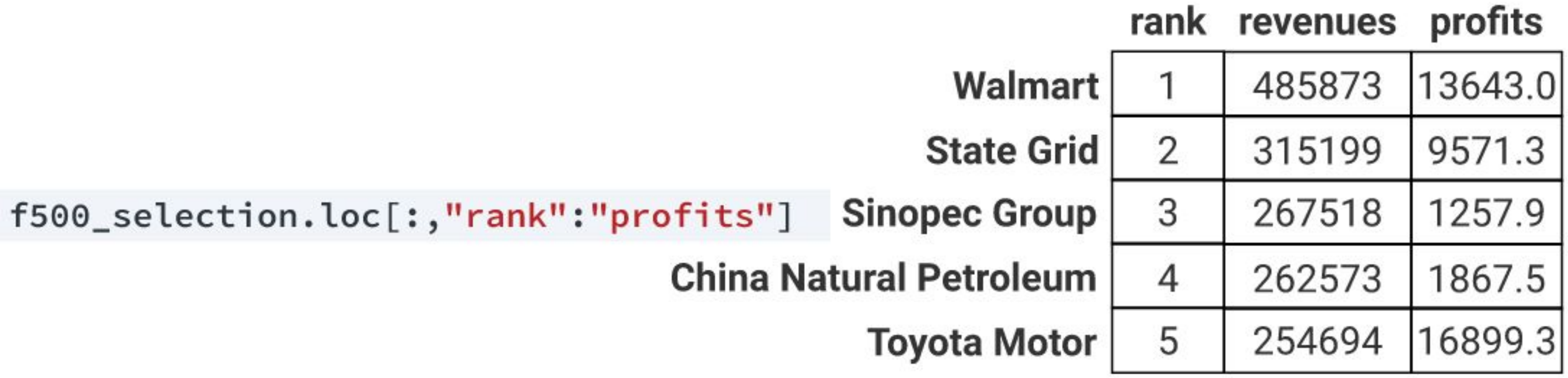
Selecting Columns (by Label) - Dataframe

```
>>> df.loc[row_label, column_label]
```

Single column: `df.loc[:,"col1"]` OR `df["col1"]`

Multiple columns: `df.loc[:,["col1","col2"]]` OR `df[["col1", "col2"]]`

Slice of columns: `df.loc[:,"col1":"coln"]` No shorthand



```
f500_selection.loc[:,"rank":"profits"]
```

	rank	revenues	profits
Walmart	1	485873	13643.0
State Grid	2	315199	9571.3
Sinopec Group	3	267518	1257.9
China Natural Petroleum	4	262573	1867.5
Toyota Motor	5	254694	16899.3

Selecting Rows (by Label) - Dataframe

```
>>> df.loc[row_label, column_label]
```

Single row: `df.loc["row1"]`

Multiple rows: `df.loc[["row1", "row2"]]`

Slice of rows: `df.loc["row1":"rown"]`

Selecting items (by Label) - Series

```
>>> df.loc[row_label, column_label]
```

Single item: `s.loc["item8"]` OR `s["item8"]`

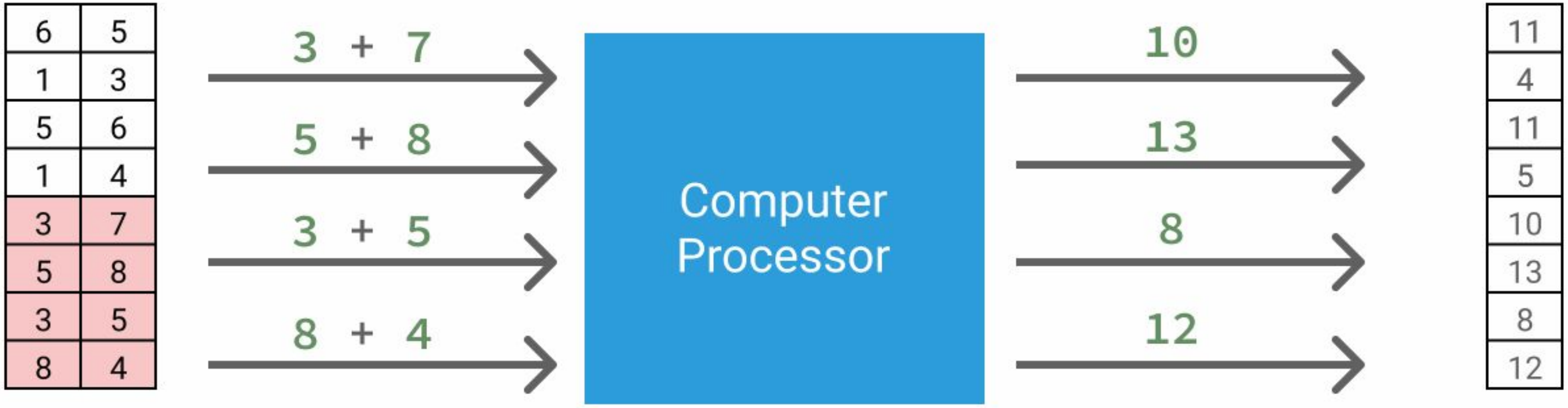
Multiple rows: `s.loc[["item1", "item7"]]` OR `s[["item1", "item7"]]`

Slice of rows: `s.loc["item2":"item4"]` OR `s["item2":"item4"]]`

Vectorization (Pandas)

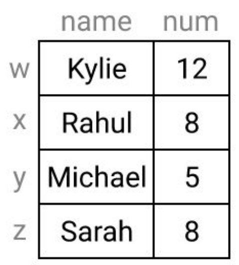
Because pandas is an extension of NumPy, it also supports vectorized operations. Let's look at an example of how this would work with a pandas series:

Just like with NumPy, we can use any of the standard Python numeric operators with series, including: `+, -, *, /`



Boolean Indexing

Let's look how boolean indexing works in pandas. For our example, we'll work with this dataframe of people and their favorite numbers:

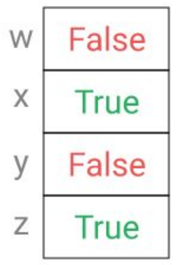


	name	num
w	Kylie	12
x	Rahul	8
y	Michael	5
z	Sarah	8

df

Let's check which people have a favorite number of **8**.

```
num_bool = df["num"] == 8
```



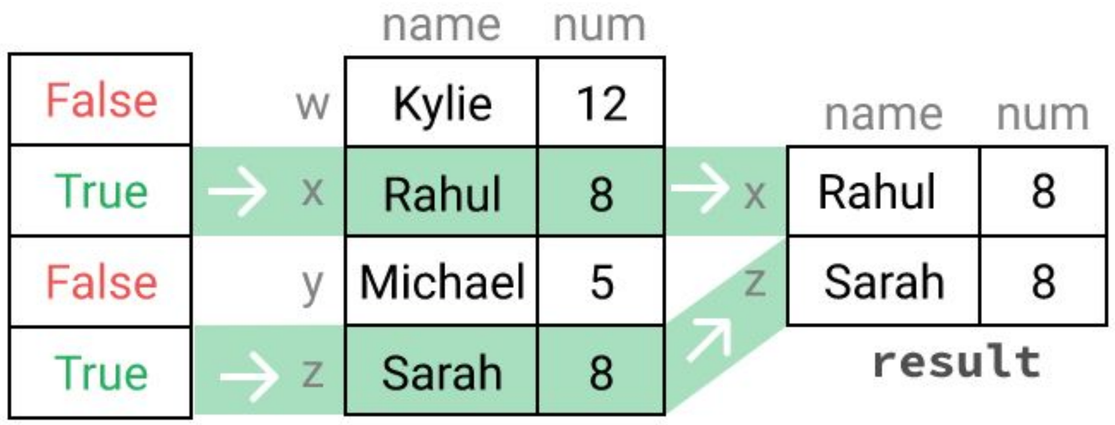
w	False
x	True
y	False
z	True

num_bool

Boolean Indexing (Cont'd)

We can use that series to index the whole dataframe, leaving us with the rows that correspond only to people whose favorite number is 8:

```
result = df[num_bool]
```



Assignment Using Boolean Arrays

First, we create a boolean series by comparing the values in the sector column to 'Motor Vehicles & Parts'

```
>>> ampersand_bool = f500["sector"] == "Motor Vehicles & Parts"
```

Next, we use that boolean series and the string "sector" to perform the assignment.

```
>>> f500.loc[ampersand_bool,"sector"] = "Motor Vehicles and Parts"
```

Just like we saw in the NumPy lesson earlier in this course, we can remove the intermediate step of creating a boolean series, and combine everything into one line. This is the most common way to write pandas code to perform assignment using boolean arrays:

```
>>> f500.loc[f500["sector"] == "Motor Vehicles & Parts","sector"] =
"Motor Vehicles and Parts"
```

Practice

Scientific Computing in Python

High performance package for data science

Intro to Pandas

Any questions?

THANKS