Introduction to NumPy: Takeaways

Syntax

SELECTING ROWS, COLUMNS, AND ITEMS FROM AN NDARRAY

- Convert a list of lists into a ndarray:

```
import numpy as np

f = open("nyc_taxis.csv", "r")

taxi_list = list(csv.reader(f))

taxi= np.array(converted_taxi_list)
```

- Selecting a row from an ndarray:

```
second_row = taxi[1]
```

- Selecting multiple rows from an ndarray:

```
all_but_first_row = taxi[1:]
```

- Selecting a specific item from an ndarray:

```
fifth_row_second_column = taxi[4,1]
```

SLICING VALUES FROM AN NDARRAY

- Selecting a single column:

```
second_column = taxi[:,1]
```

- Selecting multiple columns:

```
second_third_columns = taxi[:,1:3]

cols = [1,3,5]

second_fourth_sixth_columns = taxi[:, cols]
```

- Selecting a 2D slice:

```
twod_slice = taxi[1:4, :3]
```

VECTOR MATH

- `vector a + vector b` : Addition
- `vector a - vector b` : Subtraction
- `vector a * vector b` : Multiplication (this is unrelated to the vector multiplication used in linear algebra).
- `vector a / vector b` : Division

CALCULATING STATISTICS FOR 1D NDARRAYS

- `ndarray.min()` to calculate the minimum value
- `ndarray.max()` to calculate the maximum value
- `ndarray.mean()` to calculate the mean average value
- `ndarray.sum()` to calculate the sum of the values

CALCULATING STATISTICS FOR 2D NDARRAYS

- Max value for an entire 2D Ndarray:

```
taxi.max()
```

- Max value for each row in a 2D Ndarray (returns a 1D Ndarray):

```
taxi.max(axis=1)
```

- Max value for each column in a 2D Ndarray (returns a 1D Ndarray):

```
taxi.max(axis=0)
```

Concepts

- Python is considered a high-level language because we don't have to manually allocate memory or specify how the CPU performs certain operations. A low-level language like C gives us this control and lets us improve specific code performance, but a tradeoff in programmer productivity is made. The NumPy library lets us write code in Python but take advantage of the performance that C offers. One way NumPy makes our code run quickly is **vectorization**, which takes advantage of **Single Instruction Multiple Data (SIMD)** to process data more quickly.

- A list in NumPy is called a 1D Ndarray and a list of lists is called a 2D Ndarray. NumPy ndarrays use indices along both rows and columns and is the primary way we select and slice values.

Resources

- Arithmetic functions from the NumPy documentation.
- NumPy ndarray documentation

Boolean Indexing with NumPy: Takeaways

Syntax

READING CSV FILES WITH NUMPY

- Reading in a CSV file:

```
import numpy as np

taxi = np.genfromtxt('nyctaxis.csv', delimiter=',', skip_header=1)
```

BOOLEAN ARRAYS

- Creating a Boolean array from filtering criteria:

```
np.array([2,4,6,8]) < 5
```

- Boolean filtering for 1D ndarray:

```
a = np.array([2,4,6,8])

filter = a < 5

a[filter]
```

- Boolean filtering for 2D ndarray:

```
tip_amount = taxi[:,12]

tip_bool = tip_amount > 50

top_tips = taxi[tip_bool, 5:14]
```

ASSIGNING VALUES

- Assigning values in a 2D ndarray using indices:

```
taxi[28214,5] = 1

taxi[:,0] = 16

taxi[1800:1802,7] = taxi[:,7].mean()
```

- Assigning values using Boolean arrays:

```
taxi[taxi[:, 5] == 2, 15] = 1
```

Concepts

- Selecting values from a ndarray using Boolean arrays is very powerful. Using Boolean arrays helps us think in terms of filters on the data, instead of specific index values (like we did when working with Python lists).

Resources

- Reading a CSV file into NumPy
- Indexing and selecting data

Introduction to pandas: Takeaways

Syntax

PANDAS DATAFRAME BASICS

- Reading a file into a dataframe:

```
f500 = pd.read_csv('f500.csv',index_col=0)
```

- Returning a dataframe's data types:

```
col_types = f500.dtypes
```

- Returning the dimensions of a dataframe:

```
dims = f500.shape
```

SELECTING VALUES FROM A DATAFRAME

- Selecting a single column:

```
f500["rank"]
```

- Selecting multiple columns:

```
f500[["country", "rank"]]
```

- Selecting the first n rows:

```
first_five = f500.head(5)
```

- Selecting rows from a dataframe by label:

```
drink_companies = f500.loc[["Anheuser-Busch InBev", "Coca-Cola", "Heineken Holding"]]

big_movers = f500.loc[["Aviva", "HP", "JD.com", "BHP Billiton"], ["rank","previous_rank"]]

middle_companies = f500.loc["Tata Motors":"Nationwide", "rank":"country"]
```

Concepts

- NumPy provides fundamental structures and tools that make working with data easier, but there are several things that limit its usefulness as a single tool when working with data:
    - The lack of support for column names forces us to frame the questions we want to answer 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 lots of low-level methods ... however, there are many common analysis

    - There are lots of low level methods — however, 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 so much a replacement for NumPy as an *extension* of NumPy. The underlying code for pandas uses the NumPy library extensively. The main objects in pandas are **Series** and **Dataframes**. Series is equivalent to a 1D Ndarray while a dataframe is equivalent to a 2D Ndarray.
- Different label selection methods:

Select by Label	Explicit Syntax	Shorthand Convention
Single column from dataframe	df.loc[:,"col1"]	df["col1"]
List of columns from dataframe	df.loc[:,["col1","col7"]]	df[["col1","col7"]]
Slice of columns from dataframe	df.loc[:,"col1":"col4"]	
Single row from dataframe	df.loc["row4"]	
List of rows from dataframe	df.loc[["row1", "row8"]]	
Slice of rows from dataframe	df.loc["row3":"row5"]	df["row3":"row5"]
Single item from series	s.loc["item8"]	s["item8"]
List of items from series	s.loc[["item1","item7"]]	s[["item1","item7"]]
Slice of items from series	s.loc["item2":"item4"]	s["item2":"item4"]

Resources

- Dataframe.loc[]

- Indexing and Selecting Data

Exploring Data with pandas: Fundamentals: Takeaways

Syntax

DATA EXPLORATION METHODS

- Describing a series object:

```
revs = f500["revenues"]

summary_stats = revs.describe()
```

- Unique value counts for a column:

```
country_freqs = f500['country'].value_counts()
```

ASSIGNMENT WITH PANDAS

- Creating a new column:

```
top5_rank_revenue["year_founded"] = 0
```

- Replacing a specific value in a dataframe:

```
f500.loc["Dow Chemical","ceo"] = "Jim Fitterling"
```

BOOLEAN INDEXING IN PANDAS

- Filtering a dataframe down on a specific value in a column:

```
kr_bool = f500["country"] == "South Korea"

top_5_kr = f500[kr_bool].head()
```

- Updating values using Boolean filtering:

```
f500.loc[f500["previous_rank"] == 0, "previous_rank"] = np.nan

prev_rank_after = f500["previous_rank"].value_counts(dropna=False).head()
```

Concepts

- Because pandas is designed to operate like NumPy, a lot of concepts and methods from Numpy are supported. Examples include vectorized operations, methods for calculating summary statistics, and boolean indexing.

- **Method chaining** is a way to combine multiple methods together in a single line. When writing code, you should always assess whether method chaining will make your code harder to read. If it does, it's always preferable to break the code into more than one line.
- Because series and dataframes are two distinct objects, they have their own unique methods. However, there are many times where both series and dataframe objects have a method of the same name that behaves in similar ways.

Resources

- Series Statistics Methods
- Dataframe Statistics Methods
- Boolean Indexing

Exploring Data with pandas: Intermediate: Takeaways

Syntax

USING ILOC[] TO SELECT BY INTEGER POSITION

- Selecting a value:

```
third_row_first_col = df.iloc[2,0]
```

- Selecting a row:

```
second_row = df.iloc[1]
```

CREATING BOOLEAN MASKS USING PANDAS METHODS

- Selecting only null values in a column:

```
rev_is_null = f500["revenue_change"].isnull()
```

- Filtering using Boolean series object:

```
rev_change_null = f500[rev_is_null]
```

- Selecting only the non-null values in a column:

```
f500[f500["previous_rank"].notnull()]
```

BOOLEAN OPERATORS

- Multiple required filtering criteria:

```
filter_big_rev_neg_profit = (f500["revenues"] > 100000) & (f500["profits"] < 0)
```

- Multiple optional filtering criteria:

```
filter_big_rev_neg_profit = (f500["revenues"] > 100000) | (f500["profits"] < 0)
```

Concepts

- To select values by axis labels, use `loc[]`. To select values by integer locations, use `iloc[]`. When the label for an axis is just its integer position, these methods can be mostly used interchangeably.
- Because using a loop doesn't take advantage of vectorization, it's important to avoid doing so unless you absolutely have to. Boolean operators are a powerful technique to take advantage of vectorization when filtering because you're able to express more granular filters.

Resources

- Boolean Indexing
- iloc vs loc

Data Cleaning Basics: Takeaways

Syntax

READING A CSV IN WITH A SPECIFIC ENCODING

- Reading in a CSV file using Latin encoding:

```
laptops = pd.read_csv('laptops.csv', encoding='Latin-1')
```

- Reading in a CSV file using UTF-8:

```
laptops = pd.read_csv('laptops.csv', encoding='UTF-8')
```

- Reading in a CSV file using Windows-1251:

```
laptops = pd.read_csv('laptops.csv', encoding='Windows-1251')
```

MODIFYING COLUMNS IN A DATAFRAME

- Renaming An Existing Column:

```
laptops.rename(columns={'MANUfacturer'    'manufacturer'}, inplace=True)
```

- Converting A String Column To Float:

```
laptops["screen_size"] = laptops["screen_size"].str.replace('"','').astype(float)
```

- Converting A String Column To Integer:

```
laptops["ram"] = laptops["ram"].str.replace('GB','')

laptops["ram"] = laptops["ram"].astype(int)
```

STRING COLUMN OPERATIONS

- Extracting Values From Strings:

```
laptops["gpu_manufacturer"] = (laptops["gpu"]
                               .str.split()
                               .str[0]
```

FIXING VALUES

- Replacing Values Using A Mapping Dictionary:

```
mapping_dict = {
'Android': 'Android',
'Chrome OS': 'Chrome OS',
'Linux': 'Linux',
'Mac OS': 'macOS',
'No OS': 'No OS',
'Windows': 'Windows',
'macOS': 'macOS'
}
laptops["os"] = laptops["os"].map(mapping_dict)
```

- Dropping Missing Values:

```
laptops_no_null_rows = laptops.dropna(axis=0)
```

EXPORTING CLEANED DATA

- Exporting Cleaned Data:

```
df.to_csv("laptops_cleaned.csv", index=False)
```

Concepts

- Computers, at their lowest levels, can only understand binary.
- Encodings are systems for representing all other values in binary so a computer can work with them.
- UTF-8 is the most common encoding and is very friendly to work with in Python 3.
- When converting text data to numeric data, we usually follow the following steps:
    - Explore the data in the column.
    - Identify patterns and special cases.
    - Remove non-digit characters.
    - Convert the column to a numeric dtype.
    - Rename column if required.

Resources

- Python Encodings
- Indexing and Selecting Data