Pandas, Numpy, and List Comprehensions

Analyze data and solve maths problems

John Karuitha,PhD

March 13, 2025

Python Data Structures: Dictionaries and Sets

1. Dictionaries in Python

What is a Dictionary?

A **dictionary** in Python is an unordered collection of **key-value pairs**. It allows fast lookups and modifications based on unique keys.

Creating a Dictionary

```
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 25,
    "course": "Mathematics"
}

print(student)
```

- Keys must be **unique** and **immutable** (strings, numbers, tuples).
- Values can be **any data type**.

Accessing Values in a Dictionary

```
# Accessing values
print(student["name"])  # Alice

# Using get() method (avoids KeyError if key is missing)
print(student.get("age"))  # 25
print(student.get("grade", "Not available"))  # Default value if key is missing
```

Adding and Updating Elements

```
# Adding a new key-value pair
student["grade"] = "A"

# Updating an existing key
student["age"] = 26

print(student)
```

Removing Elements from a Dictionary

```
# Removing a specific key
del student["course"]

# Using pop() method
age = student.pop("age")  # Returns the removed value

# Removing the last inserted item (Python 3.7+)
last_item = student.popitem()  # Returns (key, value)

print(student)
```

Looping Through a Dictionary

```
# Iterating through keys
for key in student:
    print(key, ":", student[key])

# Using items() to get key-value pairs
for key, value in student.items():
    print(f"{key}: {value}")

# Getting only keys or values
print(student.keys())   # dict_keys(['name', 'grade'])
print(student.values()) # dict_values(['Alice', 'A'])
```

Dictionary Methods

```
# Copying a dictionary
student_copy = student.copy()

# Merging two dictionaries
new_data = {"hobby": "Reading", "city": "New York"}
student.update(new_data)

# Checking if a key exists
if "name" in student:
    print("Key exists!")

# Clearing all elements
student.clear()
```

---

2. Sets in Python

What is a Set?

A **set** is an unordered collection of **unique elements**. It is useful for eliminating duplicates and performing mathematical operations like **union, intersection, and difference**.

Creating a Set

```
# Creating a set
fruits = {"apple", "banana", "cherry"}
print(fruits)

# Creating an empty set (must use set(), not {})
empty_set = set()
```

Adding and Removing Elements in a Set

```
# Adding elements
fruits.add("orange")

# Removing an element (raises KeyError if not found)
fruits.remove("banana")

# Using discard() (no error if element is missing)
fruits.discard("mango")

# Popping an arbitrary element
removed_item = fruits.pop()

print(fruits)
```

Set Operations (Union, Intersection, Difference, Symmetric Difference)

Union (Combining Elements from Both Sets)

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

union_set = set_a.union(set_b)
print(union_set)  # {1, 2, 3, 4, 5}
```

Intersection (Common Elements in Both Sets)

```
intersection_set = set_a.intersection(set_b)
print(intersection_set)  # {3}
```

Difference (Elements in One Set but Not the Other)

```
difference_set = set_a.difference(set_b)
print(difference_set)  # {1, 2}
```

Symmetric Difference (Elements Not in Both Sets)

```
symmetric_difference_set = set_a.symmetric_difference(set_b)
print(symmetric_difference_set)  # {1, 2, 4, 5}
```

Checking Subsets and Supersets

```
set_x = {1, 2}
set_y = {1, 2, 3, 4}

print(set_x.issubset(set_y))  # True
print(set_y.issuperset(set_x))  # True
```

Looping Through a Set

```
for fruit in fruits:
    print(fruit)
```

Set Methods

```
# Copying a set
new_set = fruits.copy()

# Clearing all elements
fruits.clear()
```

Conclusion

Feature	Dictionary	Set
Structure	Key-Value Pairs	Unique Elements
Ordered?	No (Python 3.6+ maintains insertion order)	No
Duplicate Values?	No duplicate keys	No duplicates allowed
Mutable?	Yes	Yes
Use Case	Fast lookups, data mapping	Unique items, mathematical operations

Both **dictionaries** and **sets** are powerful data structures that help organize and process data efficiently in Python.

PANDAS (import pandas as pd)

1. Reading Files with Pandas

Pandas makes it easy to import and work with various types of files. One of the most common file formats is CSV.

Reading a CSV File:

To read a CSV file, you use the `read_csv()` function. This function converts the CSV file into a Pandas DataFrame, which is a table-like structure.

```
import pandas as pd

# Reading a CSV file
df = pd.read_csv('file_path.csv')
```

Common Options:

- **`header`**: Specify the row to use as the column names (default is the first row).
- **`index_col`**: Use a column as the index for the DataFrame.
- **`dtype`**: Specify the data type of certain columns.
- **`na_values`**: Define additional strings to recognize as missing values.

Example:

```
df = pd.read_csv('file_path.csv', header=0, index_col='ID', dtype={'Age': float}, na_values=
```

---

2. Renaming Files and Columns

Sometimes you may need to rename files or the columns in your DataFrame.

Renaming Columns:

You can rename columns by passing a dictionary of old column names as keys and new names as values.

```
# Renaming columns
df.rename(columns={'OldName': 'NewName', 'Age': 'Age_in_years'}, inplace=True)
```

Renaming Index/Rows:

You can rename the index using the `rename()` function on the DataFrame.

```
# Renaming index (rows)
df.rename(index={0: 'Row1', 1: 'Row2'}, inplace=True)
```

---

3. Selecting Rows and Columns

Selecting Columns:

You can access columns in a DataFrame using the column name.

```
# Selecting a single column
df['ColumnName']

# Selecting multiple columns
df[['Column1', 'Column2']]
```

Selecting Rows:

You can select rows by their index position or label.

- **Using `iloc` (index position):**

```
# Selecting rows by index position
df.iloc[0]  # First row
df.iloc[0:3]  # First 3 rows
df.iloc[0, 1]  # Row 1, Column 2
```

- **Using `loc` (index label):**

```
# Selecting rows by index label
df.loc[1]  # Row with label 1
df.loc[1:3]  # Rows from index 1 to 3 (inclusive)
df.loc[1, 'Column1']  # Row with label 1, Column1
```

---

4. Handling Missing Values

Missing values are a common issue in data, and Pandas provides a lot of functionality to handle them.

Identifying Missing Values:

You can check for missing values using `isnull()` or `notnull()`.

```
# Checking for missing values
df.isnull()  # Returns a DataFrame of boolean values
df.isnull().sum()  # Total missing values per column
```

Filling Missing Values:

You can replace missing values with specific values using `fillna()`.

```
# Filling missing values with a constant
df['Column'].fillna(0, inplace=True)

# Filling missing values with the mean of the column
df['Column'].fillna(df['Column'].mean(), inplace=True)
```

Dropping Missing Values:

If you want to remove rows or columns with missing values, you can use `dropna()`.

```
# Dropping rows with missing values
df.dropna(inplace=True)

# Dropping columns with missing values
df.dropna(axis=1, inplace=True)
```

---

5. Basic Data Analysis and Summary Statistics

Descriptive Statistics:

Pandas provides several methods to compute summary statistics.

```
# Summary statistics for numerical columns
df.describe()

# Summary statistics for categorical columns
df['CategoryColumn'].value_counts()
```

Correlation:

You can calculate the correlation between numerical columns.

```
# Correlation matrix
df.corr()
```

Aggregating Data:

You can group data and compute aggregate statistics like sum, mean, etc.

```
# Grouping data by a column and calculating mean
df.groupby('CategoryColumn')['ValueColumn'].mean()

# Multiple aggregation functions
df.groupby('CategoryColumn').agg({'ValueColumn': ['mean', 'sum', 'std']})
```

Sorting:

Sorting the data can help in analyzing it.

```
# Sorting by a column
df.sort_values(by='ColumnName', ascending=False, inplace=True)
```

Filtering Data:

Filtering helps in isolating relevant data for analysis.

```
# Filtering rows based on condition
df[df['ColumnName'] > 50]

# Filtering rows with multiple conditions
df[(df['ColumnName'] > 50) & (df['AnotherColumn'] == 'Value')]
```

Plotting:

You can visualize the data using Pandas' built-in plotting functionality (requires Matplotlib).

```
# Plotting a histogram
df['Column'].hist()

# Plotting a line graph
df['Column'].plot(kind='line')
```

---

6. Working with Dates

Many times, your data will have date columns, and Pandas provides robust tools to work with them.

Converting to Datetime:

You can convert a column to datetime format using `to_datetime()`.

```
df['DateColumn'] = pd.to_datetime(df['DateColumn'])
```

Extracting Date Components:

```
# Extracting the year, month, and day
df['Year'] = df['DateColumn'].dt.year
df['Month'] = df['DateColumn'].dt.month
df['Day'] = df['DateColumn'].dt.day
```

Filtering by Date:

```
# Filtering rows where the date is after a certain date
df[df['DateColumn'] > '2020-01-01']
```

---

7. Merging and Joining DataFrames

When working with multiple DataFrames, you often need to combine them.

Merging:

```
# Merging two DataFrames on a common column
df1.merge(df2, on='CommonColumn')
```

Concatenating:

```
# Concatenating DataFrames along rows
pd.concat([df1, df2], axis=0)

# Concatenating DataFrames along columns
pd.concat([df1, df2], axis=1)
```

Conclusion

These notes cover the basics to intermediate level of Pandas. Understanding how to read files, work with data, handle missing values, analyze data, and use common techniques like merging and plotting will give you a solid foundation in working with data in Python.

The next step would be to apply these concepts to real datasets, which will help solidify your understanding of Pandas.

List Comprehensions

List comprehensions in Python provide a concise way to create lists. They are often more readable and efficient compared to using traditional for-loops. Here's a breakdown of how to use list comprehensions effectively.

Basic Structure of List Comprehension

The general syntax of a list comprehension is:

```
[expression for item in iterable if condition]
```

- **`expression`**: The value to add to the list.
- **`item`**: The variable representing each element in the iterable.
- **`iterable`**: The collection (e.g., list, range, string) you are iterating over.
- **`condition`** (optional): A filter to include items that meet a certain condition.

1. Simple List Comprehensions

Create a new list by iterating over an existing iterable and applying an expression.

```
# Square of each number in a list
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

2. List Comprehensions with Condition

You can add a condition to the list comprehension to filter out elements that don't meet the condition.

```
# List of even numbers
numbers = [1, 2, 3, 4, 5, 6]
evens = [n for n in numbers if n % 2 == 0]
print(evens)  # Output: [2, 4, 6]
```

3. Nested List Comprehensions

You can nest list comprehensions to create lists of lists or perform more complex operations.

```
# Flattening a 2D list (list of lists)
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [item for sublist in matrix for item in sublist]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6]
```

4. Applying Functions in List Comprehensions

You can apply functions or perform calculations directly in the list comprehension.

```
# Convert all strings in a list to uppercase
words = ['apple', 'banana', 'cherry']
upper_words = [word.upper() for word in words]
print(upper_words)  # Output: ['APPLE', 'BANANA', 'CHERRY']
```

5. List Comprehension with Multiple Conditions

You can add more than one condition to filter items further.

```
# Numbers divisible by 2 and 3
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
divisible_by_2_and_3 = [n for n in numbers if n % 2 == 0 and n % 3 == 0]
print(divisible_by_2_and_3)  # Output: [6]
```

6. List Comprehension with `else` Statement

You can use an `else` statement to assign values when the condition is not met.

```
# List comprehension with an if-else
numbers = [1, 2, 3, 4, 5]
result = ['even' if n % 2 == 0 else 'odd' for n in numbers]
print(result)  # Output: ['odd', 'even', 'odd', 'even', 'odd']
```

7. Dictionary Comprehensions (Bonus)

List comprehensions can be extended to other data structures like dictionaries. The syntax for dictionary comprehensions is similar to list comprehensions but uses key-value pairs.

```
# Creating a dictionary where the key is the number and the value is the square
numbers = [1, 2, 3, 4, 5]
squared_dict = {n: n**2 for n in numbers}
print(squared_dict)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

8. Set Comprehensions (Bonus)

Similarly to dictionaries, you can create sets using set comprehensions.

```
# Create a set of squares (no duplicates)
numbers = [1, 2, 2, 3, 4, 5]
squares_set = {n**2 for n in numbers}
print(squares_set)  # Output: {1, 4, 9, 16, 25}
```

Summary

List comprehensions are a powerful feature in Python that can make your code shorter and more readable. Key points to remember: - They can be used for simple transformations. - Conditions and multiple loops can be added to filter and manipulate the data. - List, dictionary, and set comprehensions offer flexibility across different data types.

With practice, you'll find that list comprehensions become an essential tool in your Python toolkit, especially when working with larger datasets or more complex logic.

NUMPY (import numpy as np)

Here's a comprehensive guide on **NumPy**, which is one of the most widely used libraries for numerical computing in Python. This guide will cover key features of NumPy, from creating arrays to performing mathematical operations, linear algebra, and more. By the end of this, you'll have the foundational knowledge to work with NumPy efficiently, ranging from beginner to intermediate-level usage.

---

1. Introduction to NumPy

NumPy (Numerical Python) provides a powerful N-dimensional array object (`ndarray`) and a set of functions to operate on these arrays. It is the foundation for most numerical computations in Python and is essential for working with large datasets and performing advanced mathematical computations.

To use NumPy, you first need to import it:

```
import numpy as np
```

---

2. Creating NumPy Arrays

Creating a 1D Array:

You can create a 1-dimensional array using `np.array()` by passing a list of numbers.

```
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]
```

Creating a 2D Array:

To create a 2D array, pass a list of lists.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Creating Arrays with Built-in Functions:

- **`np.zeros()`**: Creates an array filled with zeros.

```
zeros_array = np.zeros((3, 3))  # 3x3 matrix of zeros
print(zeros_array)
```

- **`np.ones()`**: Creates an array filled with ones.

```
ones_array = np.ones((2, 4))  # 2x4 matrix of ones
print(ones_array)
```

- **`np.arange()`**: Creates an array with a range of values, similar to Python's `range()` function.

```
arr_range = np.arange(0, 10, 2)  # Values from 0 to 10, with a step of 2
print(arr_range)  # Output: [0 2 4 6 8]
```

- **`np.linspace()`**: Creates an array with evenly spaced numbers over a specified range.

```
arr_linspace = np.linspace(0, 10, 5)  # 5 evenly spaced numbers from 0 to 10
print(arr_linspace)  # Output: [ 0.   2.5  5.   7.5 10. ]
```

---

3. Basic Array Operations

Element-wise Operations:

NumPy arrays support element-wise arithmetic operations, which means you can apply arithmetic operations on arrays directly without the need for loops.

```
arr = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

# Addition
print(arr + arr2)  # Output: [ 6  8 10 12]

# Subtraction
print(arr - arr2)  # Output: [-4 -4 -4 -4]

# Multiplication
print(arr * arr2)  # Output: [ 5 12 21 32]

# Division
print(arr / arr2)  # Output: [0.2 0.33333333 0.42857143 0.5]
```

Array Broadcasting:

NumPy supports **broadcasting**, which allows operations between arrays of different shapes. The smaller array is "broadcast" over the larger one to make the shapes compatible.

```
arr = np.array([1, 2, 3, 4])
scalar = 2

# Scalar multiplication (broadcasting)
result = arr * scalar
print(result)  # Output: [2 4 6 8]
```

Array Aggregation Functions:

NumPy provides several functions to perform mathematical operations on entire arrays or along specific axes.

```python
arr = np.array([1, 2, 3, 4, 5])

# Sum
print(np.sum(arr))  # Output: 15

# Mean
print(np.mean(arr))  # Output: 3.0

# Median
print(np.median(arr))  # Output: 3.0

# Standard Deviation
print(np.std(arr))  # Output: 1.4142135623730951

# Min and Max
print(np.min(arr))  # Output: 1
print(np.max(arr))  # Output: 5
```

---

4. Array Indexing and Slicing

Indexing:

You can access elements of a NumPy array just like Python lists, but with more powerful features.

```python
arr = np.array([10, 20, 30, 40, 50])

# Accessing the 2nd element (index starts from 0)
print(arr[1])  # Output: 20
```

Slicing:

You can slice arrays using `start:stop:step` syntax.

```python
arr = np.array([10, 20, 30, 40, 50])

# Slicing from index 1 to 3 (excluding 3)
print(arr[1:3])  # Output: [20 30]
```

```
# Slicing with step
print(arr[::2])  # Output: [10 30 50]
```

Multidimensional Array Indexing:

You can index multidimensional arrays similarly, using a comma `,` to separate dimensions.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# Access element at row 1, column 2
print(arr2d[1, 2])  # Output: 6

# Slice the second row
print(arr2d[1, :])  # Output: [4 5 6]
```

---

5. Reshaping and Modifying Arrays

Reshape Arrays:

You can change the shape of a NumPy array using the `reshape()` function.

```
arr = np.array([1, 2, 3, 4, 5, 6])

# Reshape it to a 2x3 array
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Flatten Arrays:

If you want to flatten a multi-dimensional array into a 1D array, use `flatten()`.

```
arr2d = np.array([[1, 2], [3, 4], [5, 6]])

# Flatten the array
flattened_arr = arr2d.flatten()
print(flattened_arr)  # Output: [1 2 3 4 5 6]
```

Transpose:

Transpose swaps the rows and columns of a 2D array.

```
arr2d = np.array([[1, 2], [3, 4]])

# Transpose the array
transposed = arr2d.T
print(transposed)
# Output:
# [[1 3]
#  [2 4]]
```

---

6. Random Number Generation with NumPy

NumPy has a powerful suite of functions for generating random numbers.

```
# Random integer between 0 and 10
rand_int = np.random.randint(0, 10, size=5)
print(rand_int)  # Output: [7 1 4 3 9]

# Random float between 0 and 1
rand_float = np.random.random(5)
print(rand_float)  # Output: [0.21568724 0.92043762 0.98427353 0.49150107 0.42970307]

# Random values from a normal distribution
rand_normal = np.random.randn(3, 3)  # Mean = 0, Std = 1
print(rand_normal)
```

---

7. Linear Algebra with NumPy

NumPy offers many linear algebra operations such as matrix multiplication, dot products, eigenvalues, etc.

Dot Product:

```python
a = np.array([1, 2])
b = np.array([3, 4])

# Dot product of two vectors
dot_product = np.dot(a, b)
print(dot_product)  # Output: 11
```

Matrix Multiplication:

```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication
matrix_product = np.matmul(A, B)
print(matrix_product)
# Output:
# [[19 22]
#  [43 50]]
```

Finding Eigenvalues and Eigenvectors:

```python
matrix = np.array([[4, -2], [1,  1]])

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

Conclusion

NumPy is a powerful library for numerical computing in Python. This guide covered the following key areas:

- **Creating arrays** using `array()`, `zeros()`, `ones()`, `arange()`, and `linspace()`.
- **Basic array operations** such as addition, subtraction, and multiplication.
- **Indexing and slicing** arrays, including multidimensional indexing.
- **Reshaping and modifying arrays** (reshape, flatten, transpose).

- **Random number generation** using `randint()`, `random()`, and `randn()`.
- **Linear algebra operations** including dot products, matrix multiplication, and eigenvalue/eigenvector computations.

With these fundamentals, you should now be equipped to perform numerical computations and data manipulation with NumPy!