Data Science & OOP

This guide covers everything we learned in this course.

1. NumPy: Basic Functions

NumPy is a library for numerical computing, providing efficient arrays and mathematical operations. Below are key functions for creating and manipulating arrays.

Key Functions

- **np.array()**: Creates an array from a list or tuple.
- **np.zeros()**: Creates an array of zeros with a specified shape.
- **np.ones()**: Creates an array of ones with a specified shape.
- **np.arange()**: Creates an array with a range of values.
- **np.linspace()**: Creates an array with evenly spaced values over an interval.

Example: Array Creation and Line Plot

```
import numpy as np
import matplotlib.pyplot as plt

# np.array(): Create an array from a list
array = np.array([1, 2, 3, 4, 5])
print("Array from list:", array)

# np.zeros(): Create a 2x3 array of zeros
zeros = np.zeros((2, 3))
print("\n2x3 Array of zeros:\n", zeros)

# np.ones(): Create a 1x4 array of ones
ones = np.ones(4)
print("\n1x4 Array of ones:", ones)

# np.arange(): Create array from 0 to 9
range_array = np.arange(10)
print("\nRange array:", range_array)

# np.linspace(): Create 5 evenly spaced values from 0 to 10
linspace_array = np.linspace(0, 10, 5)
print("\nLinspace array:", linspace_array)
```

Output:

```
Array from list: [1 2 3 4 5]
```

```
2x3 Array of zeros:
 [[0. 0. 0.]
  [0. 0. 0.]]

1x4 Array of ones: [1. 1. 1. 1.]

Range array: [0 1 2 3 4 5 6 7 8 9]

Linspace array: [ 0.   2.5  5.   7.5 10. ]
```

Saves sine_wave.png (sine wave line plot).

2. Pandas: Basic Data Handling

Pandas provides Series (1D) and DataFrame (2D) for structured data manipulation, including reading files, converting to Excel, and accessing values.

Key Functions

- **pd.read_csv()**: Reads a CSV file into a DataFrame.
- **to_excel()**: Exports a DataFrame to an Excel file.
- **loc[] and iloc[]**: Access values by label or index.
- **values**: Returns a NumPy array of DataFrame values.

Example: File Operations and Value Access

```python
import pandas as pd
import numpy as np

# Sample data (simulate a CSV file)
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Score': [85, 90, 88]
}
df = pd.DataFrame(data)

# Save to CSV (simulating a file to read)
df.to_csv('sample.csv', index=False)

# Read CSV file
df_read = pd.read_csv('sample.csv')
print("Read CSV:\n", df_read)

# Convert to Excel
df_read.to_excel('sample.xlsx', index=False)
print("\nExported to sample.xlsx")

# Access values with loc (by label)
alice_score = df_read.loc[0, 'Score']
print("\nAlice's Score (loc):", alice_score)
```

```
# Access values with iloc (by index)
second_row = df_read.iloc[1]
print("\nSecond Row (iloc):\n", second_row)

# Get all values as NumPy array
values_array = df_read.values
print("\nDataFrame as NumPy array:\n", values_array)
```

Output:

```
Read CSV:
       Name  Age  Score
0    Alice   25     85
1      Bob   30     90
2  Charlie   35     88

Exported to sample.xlsx

Alice's Score (loc): 85

Second Row (iloc):
 Name     Bob
Age       30
Score     90
Name: 1, dtype: object

DataFrame as NumPy array:
 [['Alice' 25 85]
 ['Bob' 30 90]
 ['Charlie' 35 88]]
```

Creates sample.csv and sample.xlsx.

3. Matplotlib: Plotting and Line of Best Fit

Matplotlib creates visualizations, and with NumPy, we can compute a line of best fit for scatter plots.

Key Functions

- **plt.plot()**: Creates line plots.
- **plt.scatter()**: Creates scatter plots.
- **np.polyfit()**: Fits a polynomial (e.g., line) to data.
- **np.poly1d()**: Creates a polynomial function from coefficients.

Example: Scatter Plot with Line of Best Fit

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([2.1, 3.8, 6.2, 7.9, 9.8])

# Scatter plot
plt.figure(figsize=(6, 4))
plt.scatter(x, y, color='red', label='Data Points')

# Calculate line of best fit (1st-degree polynomial)
coefficients = np.polyfit(x, y, 1)  # Linear fit
polynomial = np.poly1d(coefficients)
line_y = polynomial(x)

# Plot line of best fit
plt.plot(x, line_y, 'b-', label=f'Fit: y = {coefficients[0]:.2f}x +
{coefficients[1]:.2f}')

plt.title('Scatter Plot with Line of Best Fit')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.savefig('best_fit.png')
plt.close()
```

Output: Saves best_fit.png (scatter plot with a line of best fit).

Key Takeaways

- **NumPy**: Use np.array(), np.zeros(), np.ones(), np.arange(), and np.linspace() for array creation; supports line plots.
- **Pandas**: Read files with pd.read_csv(), export to Excel with to_excel(), access data with loc, iloc, and values.
- **Matplotlib**: Create scatter plots and compute lines of best fit with np.polyfit().

Practice these examples to master the basics for your exam!

2D Lists and Tuples in Python

This guide covers 2D lists (with code tracing, nested loops, and time complexity) and tuples (including what can and cannot be added) for your Data Structures and OOP exam preparation. Referencing is explained simply, focusing on how modifying a list affects another list assigned to it.

1. 2D Lists: Basics and Operations

A 2D list in Python is a list of lists, representing a matrix or grid where each element in the outer list is a row containing elements (columns).

Key Points

- **Structure**: matrix = [[1, 2, 3], [4, 5, 6]] (2 rows, 3 columns).
- **Accessing Elements**: Use matrix[row][col] to access or modify elements.
- **Initialization**: Create manually or with loops.
- **Applications**: Used for grids, tables, or matrices.

Example: Creating and Accessing a 2D List

```python
# Initialize a 2x3 matrix
matrix = [[1, 2, 3], [4, 5, 6]]
print("Matrix:", matrix)

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

# Modify element
matrix[0][1] = 10
print("Modified Matrix:", matrix)
```

Output:

```
Matrix: [[1, 2, 3], [4, 5, 6]]
Element at [1][2]: 6
Modified Matrix: [[1, 10, 3], [4, 5, 6]]
```

Tracing Code with 2D Lists

Tracing involves stepping through code to understand its execution. Below is a function that finds the maximum element in a 2D list.

Example: Find Maximum in 2D List

```python
def find_max_2d(matrix):
    max_value = matrix[0][0]
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            if matrix[i][j] > max_value:
                max_value = matrix[i][j]
    return max_value

# Test matrix
matrix = [[1, 2, 3], [4, 5, 6]]
result = find_max_2d(matrix)
print("Maximum value:", result)
```

Trace:

1. Input: matrix = [[1, 2, 3], [4, 5, 6]]
2. Initialization: max_value = matrix[0][0] = 1
3. Outer Loop (i):
    - i = 0: Inner loop iterates over j = 0, 1, 2
        - j = 0: matrix[0][0] = 1, 1 > 1 is false, max_value = 1
        - j = 1: matrix[0][1] = 2, 2 > 1 is true, max_value = 2
        - j = 2: matrix[0][2] = 3, 3 > 2 is true, max_value = 3
    - i = 1: Inner loop iterates over j = 0, 1, 2
        - j = 0: matrix[1][0] = 4, 4 > 3 is true, max_value = 4
        - j = 1: matrix[1][1] = 5, 5 > 4 is true, max_value = 5
        - j = 2: matrix[1][2] = 6, 6 > 5 is true, max_value = 6
4. Return: max_value = 6

Output: Maximum value: 6

Key Insight: Tracing shows how nested loops compare each element to update max_value.

Nested Loops and Time Complexity

Nested loops are used to iterate over 2D lists, processing each element.

Time Complexity of for Loops

- **Single for Loop**: O(n) where n is the number of iterations.
- **Nested Loops (2D List)**: For a matrix with m rows and n columns, the time complexity is O(m * n) because each element is visited once.
- **Special Case**: If rows and columns are equal (m = n), complexity is O(n^2).

Example: Print 2D List Elements

```
matrix = [[1, 2, 3], [4, 5, 6]]

for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        print(f"Element at [{i}][{j}]: {matrix[i][j]}")
```

Output:

```
Element at [0][0]: 1
Element at [0][1]: 2
Element at [0][2]: 3
Element at [1][0]: 4
Element at [1][1]: 5
Element at [1][2]: 6
```

Trace:

1. Step 1: i = 0, j = 0, 1, 2 → Prints matrix[0][0], matrix[0][1], matrix[0][2].
2. Step 2: i = 1, j = 0, 1, 2 → Prints matrix[1][0], matrix[1][1], matrix[1][2].

Time Complexity: O(m * n) where m = 2 (rows), n = 3 (columns), so 2 * 3 = 6 iterations.

Referencing in Lists

When you assign one list to another (lst1 = lst2), both variables reference the same list object. Modifying lst2 also modifies lst1.

Example: List Referencing

```
# Create a 2D list
lst1 = [[1, 2], [3, 4]]
lst2 = lst1  # lst2 references lst1

# Modify lst2
lst2[0][0] = 99
print("lst1 after modifying lst2:", lst1)
print("lst2:", lst2)
```

Output:

```
lst1 after modifying lst2: [[99, 2], [3, 4]]
lst2: [[99, 2], [3, 4]]
```

Trace:

1. Assignment: lst2 = lst1 makes lst2 point to the same list object as lst1.
2. Modification: lst2[0][0] = 99 changes the first element of the shared list, so lst1[0][0] also becomes 99.

Key Insight: Both lst1 and lst2 reference the same memory location, so changes to one affect the other.

2. Tuples: Immutable Sequences

Tuples are immutable sequences in Python, meaning their elements cannot be changed after creation. They are often used for fixed data or as dictionary keys.

Key Points

- **Creation**: t = (1, 2, 3) or t = tuple([1, 2, 3]).
- **What You Can Add:**
  - Tuples can be elements in a list or another tuple.
  - You can create a new tuple by concatenating tuples (e.g., t1 + t2).
  - You can assign a tuple to a list or variable.
- **What You Cannot Add:**
  - You cannot modify, add, or remove elements in an existing tuple (e.g., t[0] = 5 raises an error).

    - You cannot append or pop elements like with lists.
- **Access**: Use indexing (t[0]) or slicing (t[1:3]).

Example: Tuples and Operations

```python
# Create a tuple
t1 = (1, 2, 3)
print("Tuple t1:", t1)

# Access element
print("First element:", t1[0])  # Output: 1

# What you can add: Concatenate tuples
t2 = (4, 5)
t3 = t1 + t2
print("Concatenated tuple:", t3)

# What you can add: Tuple in a list
list_with_tuple = [t1, (6, 7)]
print("List with tuples:", list_with_tuple)

# What you cannot add: Modify tuple
try:
    t1[0] = 99  # Error: tuples are immutable
except TypeError as e:
    print("Error:", e)

# What you cannot add: Append to tuple
try:
    t1.append(4)  # Error: tuples have no append method
except AttributeError as e:
    print("Error:", e)
```

Output:

```
Tuple t1: (1, 2, 3)
First element: 1
Concatenated tuple: (1, 2, 3, 4, 5)
List with tuples: [(1, 2, 3), (6, 7)]
Error: 'tuple' object does not support item assignment
Error: 'tuple' object has no attribute 'append'
```

Trace:

1. Creation: t1 = (1, 2, 3) creates an immutable tuple.
2. Access: t1[0] retrieves 1.
3. Concatenation: t1 + t2 creates a new tuple (1, 2, 3, 4, 5).
4. List with Tuple: list_with_tuple stores t1 and (6, 7) as elements.

5. Modification Attempt: t1[0] = 99 fails because tuples are immutable.
6. Append Attempt: t1.append(4) fails because tuples lack methods like append.

Mutable Elements in Tuples

While tuples themselves are immutable, they can contain mutable elements like lists. This means you can't modify the tuple structure, but you can modify mutable elements inside it.

Example: Modifying Lists Inside Tuples

```python
# Create a tuple containing a list
t = (1, [2, 3, 4], 5)
print("Original tuple:", t)

# You can modify the list inside the tuple
t[1].append(6)
print("After modifying list:", t)

# You can also modify list elements
t[1][0] = 99
print("After modifying list element:", t)

# But you cannot modify the tuple structure
try:
    t[1] = [7, 8, 9]  # Error: can't modify tuple
except TypeError as e:
    print("Error:", e)

# You can't add new elements to the tuple
try:
    t.append(7)  # Error: tuples have no append method
except AttributeError as e:
    print("Error:", e)
```

Output:

```
Original tuple: (1, [2, 3, 4], 5)
After modifying list: (1, [2, 3, 4, 6], 5)
After modifying list element: (1, [99, 3, 4, 6], 5)
Error: 'tuple' object does not support item assignment
Error: 'tuple' object has no attribute 'append'
```

Trace:

1. Creation: t = (1, [2, 3, 4], 5) creates a tuple with a list as its second element.
2. List Modification: t[1].append(6) adds 6 to the list inside the tuple.
3. List Element Modification: t[1][0] = 99 changes the first element of the list.
4. Tuple Modification Attempt: t[1] = [7, 8, 9] fails because you can't modify the tuple structure.

5. Append Attempt: t.append(7) fails because tuples don't have an append method.

Key Insight: While the tuple itself is immutable, you can modify mutable objects (like lists) that are elements of the tuple. This is because the tuple only stores references to these objects, and the objects themselves can be modified.

Example: Nested Lists in Tuples

```
# Create a tuple with nested lists
t = ([1, 2], [3, 4, 5])
print("Original tuple:", t)

# Modify nested list
t[0].append(3)
print("After modifying first list:", t)

# Modify element in nested list
t[1][1] = 99
print("After modifying second list element:", t)

# Create a new list and try to assign it
try:
    t[0] = [7, 8, 9]  # Error: can't modify tuple
except TypeError as e:
    print("Error:", e)
```

Output:

```
Original tuple: ([1, 2], [3, 4, 5])
After modifying first list: ([1, 2, 3], [3, 4, 5])
After modifying second list element: ([1, 2, 3], [3, 99, 5])
Error: 'tuple' object does not support item assignment
```

Trace:

1. Creation: t = ([1, 2], [3, 4, 5]) creates a tuple with two lists.
2. First List Modification: t[0].append(3) adds 3 to the first list.
3. Second List Modification: t[1][1] = 99 changes the second element of the second list.
4. Tuple Modification Attempt: t[0] = [7, 8, 9] fails because you can't modify the tuple structure.

Key Insight: This demonstrates that you can modify any level of nesting in lists within tuples, but you cannot change the tuple structure itself.

Adding Elements to Tuples

While tuples are immutable, you can add elements by creating new tuples. Here are the main methods:

Example: Adding Elements to Tuples

```
# Create initial tuple
t1 = (1, 2, 3)
print("Original tuple:", t1)

# Method 1: Concatenation with + operator
t2 = t1 + (4, 5)
print("After concatenation:", t2)

# Method 2: Adding single element (note the comma)
t3 = t1 + (4,)  # The comma is important!
print("After adding single element:", t3)

# Common mistake: missing comma
try:
    t4 = t1 + (4)  # Error: can't add tuple and int
except TypeError as e:
    print("Error:", e)

# Adding to tuple with mutable elements
t5 = ([1, 2], [3, 4])
t6 = t5 + ([5, 6],)
print("After adding to tuple with lists:", t6)
```

Output:

```
Original tuple: (1, 2, 3)
After concatenation: (1, 2, 3, 4, 5)
After adding single element: (1, 2, 3, 4)
Error: can only concatenate tuple (not "int") to tuple
After adding to tuple with lists: ([1, 2], [3, 4], [5, 6])
```

Key Insights:

1. You can't modify existing tuples, but you can create new ones with additional elements.
2. When adding a single element, remember the comma: (4,) is a tuple, (4) is an integer.
3. The + operator concatenates tuples, creating a new tuple.
4. You can add any type of element, including mutable objects like lists.

Removing Elements from Tuples

Since tuples are immutable, you can't directly remove elements. Instead, you create a new tuple without the elements you want to remove.

Example: Removing Elements from Tuples

```
# Create initial tuple
t1 = (1, 2, 3, 4, 5)
```

```
print("Original tuple:", t1)

# Method 1: Using slicing to remove elements
t2 = t1[1:4]  # Remove first and last elements
print("After slicing:", t2)

# Method 2: Remove specific element by creating new tuple
t3 = t1[:2] + t1[3:]  # Remove element at index 2
print("After removing element at index 2:", t3)

# Method 3: Remove first occurrence of a value
value_to_remove = 3
t4 = t1[:t1.index(value_to_remove)] + t1[t1.index(value_to_remove)+1:]
print("After removing value 3:", t4)

# Common mistake: trying to remove directly
try:
    t1.remove(3)  # Error: tuples have no remove method
except AttributeError as e:
    print("Error:", e)
```

Output:

```
Original tuple: (1, 2, 3, 4, 5)
After slicing: (2, 3, 4)
After removing element at index 2: (1, 2, 4, 5)
After removing value 3: (1, 2, 4, 5)
Error: 'tuple' object has no attribute 'remove'
```

Key Insights:

1. You can't modify existing tuples, but you can create new ones without certain elements.
2. Use slicing to remove elements from the beginning or end.
3. To remove a specific element, combine slices before and after that element.
4. To remove a specific value, find its index and combine slices.
5. Remember that tuples don't have methods like remove() or pop().

3. Dictionaries, Nested Dictionaries, and Sets

Dictionaries

Dictionaries are mutable, unordered collections of key-value pairs. Keys must be unique and immutable (strings, numbers, or tuples), while values can be any type.

Example: Basic Dictionary Operations

```
# Create a dictionary
student = {
    'name': 'John',
    'age': 20,
    'grades': [85, 90, 88]
}
print("Original dictionary:", student)

# Access values
print("Name:", student['name'])
print("Age:", student.get('age'))  # Using get() method

# Add/Modify items
student['major'] = 'Computer Science'
student['age'] = 21
print("After modifications:", student)

# Remove items
del student['grades']
print("After removing grades:", student)

# Check if key exists
print("Has name?", 'name' in student)
print("Has grades?", 'grades' in student)

# Get all keys and values
print("Keys:", list(student.keys()))
print("Values:", list(student.values()))
```

Output:

```
Original dictionary: {'name': 'John', 'age': 20, 'grades': [85, 90, 88]}
Name: John
Age: 20
After modifications: {'name': 'John', 'age': 21, 'grades': [85, 90, 88], 'major':
'Computer Science'}
After removing grades: {'name': 'John', 'age': 21, 'major': 'Computer Science'}
Has name? True
Has grades? False
Keys: ['name', 'age', 'major']
Values: ['John', 21, 'Computer Science']
```

Example: Dictionary Iteration

```
# Create a dictionary
student = {
    'name': 'John',
    'age': 20,
    'grades': [85, 90, 88]
}
```

```
# Iterate through keys
print("Keys:")
for key in student:
    print(f"- {key}")

# Iterate through key-value pairs
print("\nKey-Value pairs:")
for key, value in student.items():
    print(f"- {key}: {value}")

# Iterate through values
print("\nValues:")
for value in student.values():
    print(f"- {value}")

# Dictionary comprehension
squares = {x: x**2 for x in range(5)}
print("\nSquares dictionary:", squares)
```

Output:

```
Keys:
- name
- age
- grades

Key-Value pairs:
- name: John
- age: 20
- grades: [85, 90, 88]

Values:
- John
- 20
- [85, 90, 88]

Squares dictionary: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Nested Dictionaries

Nested dictionaries are dictionaries that contain other dictionaries as values. They're useful for representing hierarchical data.

Example: Nested Dictionary Operations

```
# Create a nested dictionary
school = {
    'students': {
        'john': {'age': 20, 'major': 'CS'},
        'alice': {'age': 19, 'major': 'Math'}
    },
    'teachers': {
        'smith': {'subject': 'Python', 'years': 5},
        'jones': {'subject': 'Java', 'years': 3}
    }
}

# Access nested values
print("John's major:", school['students']['john']['major'])
print("Smith's subject:", school['teachers']['smith']['subject'])

# Modify nested values
school['students']['john']['age'] = 21
print("Updated John's age:", school['students']['john']['age'])

# Add new nested dictionary
school['students']['bob'] = {'age': 18, 'major': 'Physics'}
print("Added Bob:", school['students']['bob'])

# Remove nested dictionary
del school['teachers']['jones']
print("After removing Jones:", school['teachers'])
```

Output:

```
John's major: CS
Smith's subject: Python
Updated John's age: 21
Added Bob: {'age': 18, 'major': 'Physics'}
After removing Jones: {'smith': {'subject': 'Python', 'years': 5}}
```

Sets

Sets are mutable, unordered collections of unique elements. They're useful for mathematical operations and removing duplicates.

Example: Set Operations

```
# Create sets
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
print("Set 1:", set1)
print("Set 2:", set2)

# Basic operations
print("Union:", set1 | set2)  # or set1.union(set2)
print("Intersection:", set1 & set2)  # or set1.intersection(set2)
print("Difference:", set1 - set2)  # or set1.difference(set2)
print("Symmetric difference:", set1 ^ set2)  # or set1.symmetric_difference(set2)

# Add and remove elements
set1.add(6)
print("After adding 6:", set1)
set1.remove(1)
print("After removing 1:", set1)

# Check membership
print("Is 3 in set1?", 3 in set1)
print("Is 9 in set1?", 9 in set1)

# Convert list to set to remove duplicates
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
unique_numbers = set(numbers)
print("Unique numbers:", unique_numbers)
```

Output:

```
Set 1: {1, 2, 3, 4, 5}
Set 2: {4, 5, 6, 7, 8}
Union: {1, 2, 3, 4, 5, 6, 7, 8}
Intersection: {4, 5}
Difference: {1, 2, 3}
Symmetric difference: {1, 2, 3, 6, 7, 8}
After adding 6: {1, 2, 3, 4, 5, 6}
After removing 1: {2, 3, 4, 5, 6}
Is 3 in set1? True
Is 9 in set1? False
Unique numbers: {1, 2, 3, 4}
```

Key Insights:

1. Dictionaries:
    - Use key-value pairs for structured data
    - Keys must be unique and immutable
    - Values can be any type
    - Use get() for safe access
2. Nested Dictionaries:
    - Useful for hierarchical data
    - Access using multiple keys
    - Can modify/add/remove at any level
3. Sets:
    - Store unique elements
    - Support mathematical operations

  - Useful for removing duplicates
  - Fast membership testing

Example: Set Immutability Restrictions

```python
# Valid set elements (immutable)
valid_set = {1, 2, 3, 'hello', (1, 2, 3)}
print("Valid set:", valid_set)

# Invalid set elements (mutable)
try:
    invalid_set = {[1, 2, 3]}  # List is mutable
except TypeError as e:
    print("Error with list:", e)

try:
    invalid_set = {{'a': 1}}  # Dictionary is mutable
except TypeError as e:
    print("Error with dict:", e)

try:
    invalid_set = {{1, 2, 3}}  # Set is mutable
except TypeError as e:
    print("Error with set:", e)

# Tuple with mutable elements is also invalid
try:
    invalid_set = {([1, 2], 3)}  # Tuple contains mutable list
except TypeError as e:
    print("Error with tuple containing list:", e)

# Valid: tuple with only immutable elements
valid_set = {(1, 2, 3), (4, 5, 6)}
print("Valid set with tuples:", valid_set)
```

Output:

```
Valid set: {1, 2, 3, 'hello', (1, 2, 3)}
Error with list: unhashable type: 'list'
Error with dict: unhashable type: 'dict'
Error with set: unhashable type: 'set'
Error with tuple containing list: unhashable type: 'list'
Valid set with tuples: {(1, 2, 3), (4, 5, 6)}
```

Key Insights:

1. Dictionary Iteration:

    - Use for key in dict to iterate through keys

- Use .items() to get key-value pairs
- Use .values() to get only values
- Dictionary comprehensions for creating new dictionaries
- Can filter dictionaries using comprehensions

2. Set Immutability:

    - Sets can only contain immutable elements
    - Valid elements: numbers, strings, tuples (if they contain only immutable elements)
    - Invalid elements: lists, dictionaries, sets, tuples containing mutable elements
    - This is because sets use hashing for fast lookups, and mutable objects can't be hashed

4. Common Python Errors

Types of Errors

1. Syntax Errors

These occur when Python can't understand your code due to incorrect syntax.

```python
# Missing colon
if True
    print("Hello")  # SyntaxError: expected ':'

# Unmatched parentheses
print("Hello"  # SyntaxError: unexpected EOF while parsing

# Incorrect indentation
def hello():
print("Hello")  # IndentationError: expected an indented block
```

2. Runtime Errors (Exceptions)

These occur while the program is running.

```python
# TypeError: Wrong type of argument
"hello" + 5  # TypeError: can only concatenate str (not "int") to str

# ValueError: Wrong value
int("abc")  # ValueError: invalid literal for int() with base 10: 'abc'

# IndexError: List index out of range
lst = [1, 2, 3]
lst[5]  # IndexError: list index out of range

# KeyError: Dictionary key not found
d = {'a': 1}
d['b']  # KeyError: 'b'
```

```
# AttributeError: Object has no attribute
lst = [1, 2, 3]
lst.appendd(4)  # AttributeError: 'list' object has no attribute 'appendd'
```

3. Semantic Errors

These occur when the code runs but doesn't do what you intended.

```
# Wrong variable name
name = "John"
print(nmae)  # NameError: name 'nmae' is not defined

# Logic error in loop
numbers = [1, 2, 3, 4, 5]
sum = 0
for i in range(len(numbers)):
    sum = numbers[i]  # Should be sum += numbers[i]
print("Sum:", sum)  # Prints 5 instead of 15

# Incorrect condition
age = 20
if age = 18:  # Should be age == 18
    print("Adult")
```

Key Points:

1. Syntax Errors: Caught before program runs (missing colons, unmatched brackets, wrong indentation)
2. Runtime Errors: Occur during execution (TypeError, ValueError, IndexError, KeyError, AttributeError)
3. Semantic Errors: Program runs but doesn't work as intended (wrong variable names, logic errors)

5. List Comprehensions, Map, and Filter

List Comprehensions

A concise way to create lists based on existing lists.

```
# Basic list comprehension
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
print("Squares:", squares)  # [1, 4, 9, 16, 25]

# With condition
even_squares = [x**2 for x in numbers if x % 2 == 0]
print("Even squares:", even_squares)  # [4, 16]

# Nested list comprehension
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print("Flattened:", flattened)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Map Function

Applies a function to each item in an iterable.

```
# Using map with str()
numbers = [1, 2, 3, 4, 5]
strings = list(map(str, numbers))
print("As strings:", strings)  # ['1', '2', '3', '4', '5']

# Using map with int()
string_numbers = ['1', '2', '3', '4', '5']
integers = list(map(int, string_numbers))
print("As integers:", integers)  # [1, 2, 3, 4, 5]
```

Filter Function

Creates an iterator of elements for which a function returns true.

```
# Using filter with str.isdigit()
mixed = ['1', '2', 'a', '3', 'b', '4']
numbers = list(filter(str.isdigit, mixed))
print("Only digits:", numbers)  # ['1', '2', '3', '4']

# Filter with None (removes falsy values)
values = [0, 1, "", "hello", [], [1, 2], None, False]
truthy = list(filter(None, values))
print("Truthy values:", truthy)  # [1, 'hello', [1, 2]]
```

Key Points:

1. List Comprehensions: Most readable for simple transformations
2. Map: Good for converting between types (str to int, int to str)
3. Filter: Best for selecting elements based on a condition

6. File Operations and JSON

Basic File Operations

```
# Writing to a file
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()

# Reading entire file
file = open("example.txt", "r")
```

```python
content = file.read()
print("Read entire file:", content)
file.close()

# Reading line by line
file = open("example.txt", "r")
line = file.readline()
print("Read first line:", line)
file.close()

# Reading all lines
file = open("example.txt", "r")
lines = file.readlines()
print("Read all lines:", lines)
file.close()

# Writing multiple lines
file = open("example.txt", "w")
file.write("Line 1\n")
file.write("Line 2\n")
file.write("Line 3")
file.close()

# Writing from dictionary
data = {
    "name": "John",
    "age": 30,
    "city": "New York"
}
file = open("data.txt", "w")
for key, value in data.items():
    file.write(f"{key}: {value}\n")
file.close()

# Reading into dictionary
file = open("data.txt", "r")
loaded_data = {}
for line in file:
    key, value = line.strip().split(": ")
    loaded_data[key] = value
file.close()
print("Loaded data:", loaded_data)
```

Output:

```
Read entire file: Hello, World!
Read first line: Hello, World!
Read all lines: ['Hello, World!']
Loaded data: {'name': 'John', 'age': '30', 'city': 'New York'}
```

Key Points:

1. Reading Methods:
   - read(): reads entire file
   - readline(): reads one line
   - readlines(): reads all lines into list
2. Writing Methods:
   - write(): writes string to file
   - Can write multiple lines with \n
   - Can write data structures line by line
3. File Modes:
   - "w": write (overwrites)
   - "r": read
   - "a": append

JSON Operations

```python
import json
# Dictionary to JSON string
data = {
    "name": "John",
    "age": 30,
    "city": "New York"
}
json_string = json.dumps(data)
print("JSON string:", json_string)

# JSON string to dictionary
parsed_data = json.loads(json_string)
print("Parsed data:", parsed_data)

# Writing JSON to file
file = open("data.json", "w")
json.dump(data, file)
file.close()

# Reading JSON from file
file = open("data.json", "r")
loaded_data = json.load(file)
print("Loaded data:", loaded_data)
file.close()
```

Basic Exception Handling

```
# Try-Except
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print("Result:", result)
except ValueError:
    print("Please enter a valid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
except:
    print("Something went wrong")
```

Key Points:

1. File Operations:
    - "w" for writing (overwrites)
    - "r" for reading
    - "a" for appending
    - Always close files
2. JSON:
    - dumps(): dictionary to string
    - loads(): string to dictionary
    - dump(): dictionary to file
    - load(): file to dictionary
3. Exceptions:
    - Try block for code that might fail
    - Except blocks for specific errors
    - Generic except for unknown errors

String Methods: strip() and rstrip()

```
# strip() removes whitespace from both ends
text = "   Hello, World!   "
print("Original:", repr(text))
print("After strip():", repr(text.strip()))

# rstrip() removes whitespace from right end only
text = "   Hello, World!   "
print("After rstrip():", repr(text.rstrip()))

# strip() with specific characters
text = "...Hello, World!..."
print("Original:", text)
print("After strip('.'):", text.strip('.'))
```

```
# rstrip() with specific characters
text = "Hello, World!..."
print("Original:", text)
print("After rstrip('.'):", text.rstrip('.'))

# Common use: cleaning file input
file = open("example.txt", "w")
file.write("Line 1   \n")
file.write("Line 2   \n")
file.close()

file = open("example.txt", "r")
for line in file:
    print("Original line:", repr(line))
    print("After strip():", repr(line.strip()))
file.close()
```

Output:

```
Original: '   Hello, World!   '
After strip(): 'Hello, World!'
After rstrip(): '   Hello, World!'
Original: ...Hello, World!...
After strip('.'): 'Hello, World!'
Original: Hello, World!...
After rstrip('.'): 'Hello, World!'
Original line: 'Line 1   \n'
After strip(): 'Line 1'
Original line: 'Line 2   \n'
After strip(): 'Line 2'
```

Key Points:

1. strip():
    - Removes whitespace from both ends
    - Can remove specific characters from both ends
    - Useful for cleaning user input or file data
2. rstrip():
    - Removes whitespace from right end only
    - Can remove specific characters from right end
    - Useful when you want to keep leading spaces

7. Stack ADT

A Stack is a Last-In-First-Out (LIFO) data structure where elements are added and removed from the same end (top).

Basic Stack Operations

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        """Add an item to the top of the stack"""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item from the stack"""
        if not self.isEmpty():
            return self.items.pop()
        return None

    def top(self):
        """Return the top item without removing it"""
        if not self.isEmpty():
            return self.items[-1]
        return None

    def isEmpty(self):
        """Check if the stack is empty"""
        return len(self.items) == 0

    def __len__(self):
        """Return the number of items in the stack"""
        return len(self.items)

# Example usage
stack = Stack()

# Push items
stack.push(1)
stack.push(2)
stack.push(3)
print("Stack after pushes:", stack.items)  # [1, 2, 3]

# Check top
print("Top item:", stack.top())  # 3

# Pop items
print("Popped item:", stack.pop())  # 3
print("Stack after pop:", stack.items)  # [1, 2]

# Check length and emptiness
print("Stack length:", len(stack))  # 2
print("Is empty?", stack.isEmpty())  # False

# Pop all items
stack.pop()
stack.pop()
print("Is empty?", stack.isEmpty())  # True
```

Key Points:

1. LIFO Principle:
    - Last item pushed is the first item popped
    - Like a stack of plates
2. Basic Operations:
    - push(item): Add to top
    - pop(): Remove from top
    - top(): View top item
    - isEmpty(): Check if empty
    - len(): Get size
3. Common Uses:
    - Function call stack
    - Undo operations
    - Expression evaluation
    - Backtracking algorithms

8. Object-Oriented Programming

Basic Classes

```
class Person:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def greet(self) -> str:
        return f"Hello, I am {self.name}"

# Creating and using objects
person = Person("John", 30)
print(person.greet())  # Hello, I am John
```

Containment (Has-A) and Annotations

```
from typing import List

class Course:
    def __init__(self, name: str, code: str):
        self.name = name
        self.code = code

class Student:
    def __init__(self, name: str, id: int):
        self.name = name
        self.id = id
        self.courses: List[Course] = []  # Has-A relationship

    def add_course(self, course: Course) -> None:
        self.courses.append(course)

    def get_courses(self) -> List[str]:
        return [course.name for course in self.courses]

# Using containment
student = Student("Alice", 123)
math = Course("Mathematics", "MATH101")
student.add_course(math)
print(student.get_courses())  # ['Mathematics']
```

Inheritance (Is-A)

```
class Animal:
    def __init__(self, name: str):
        self.name = name

    def speak(self) -> str:
        return "Some sound"

class Dog(Animal):  # Dog is-a Animal
    def speak(self) -> str:
        return "Woof!"

class Cat(Animal):  # Cat is-a Animal
    def speak(self) -> str:
        return "Meow!"

# Using inheritance
dog = Dog("Buddy")
cat = Cat("Whiskers")
print(dog.speak())  # Woof!
print(cat.speak())  # Meow!
```

Abstract Classes

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
```

```python
    def area(self) -> float:
        pass

    @abstractmethod
    def perimeter(self) -> float:
        pass

class Rectangle(Shape):
    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def area(self) -> float:
        return self.width * self.height

    def perimeter(self) -> float:
        return 2 * (self.width + self.height)

# Using abstract class
rect = Rectangle(5, 3)
print(f"Area: {rect.area()}")  # 15
print(f"Perimeter: {rect.perimeter()}")  # 16
```

Relational Operators and Fraction Class

```python
from functools import total_ordering

@total_ordering
class Fraction:
    def __init__(self, numerator: int, denominator: int):
        if denominator == 0:
            raise ValueError("Denominator cannot be zero")
        self.numerator = numerator
        self.denominator = denominator

    def __eq__(self, other: 'Fraction') -> bool:
        return (self.numerator * other.denominator ==
                other.numerator * self.denominator)

    def __lt__(self, other: 'Fraction') -> bool:
        return (self.numerator * other.denominator <
                other.numerator * self.denominator)

    def __add__(self, other: 'Fraction') -> 'Fraction':
        new_num = (self.numerator * other.denominator +
                   other.numerator * self.denominator)
        new_den = self.denominator * other.denominator
        return Fraction(new_num, new_den)

    def __str__(self) -> str:
        return f"{self.numerator}/{self.denominator}"
```

```
# Using Fraction class
f1 = Fraction(1, 2)
f2 = Fraction(1, 3)
print(f1 + f2)  # 5/6
print(f1 < f2)  # False
print(f1 > f2)  # True (automatically provided by @total_ordering)
print(f1 <= f2)  # False (automatically provided by @total_ordering)
print(f1 >= f2)  # True (automatically provided by @total_ordering)
```

Key Points:

1. @total_ordering:
    - Only need to implement **eq** and **lt**
    - Automatically provides **gt**, **le**, **ge**
    - Makes all comparison operators work
    - More efficient than implementing all operators
2. Without @total_ordering:
    - Would need to implement all operators: **eq**, **lt**, **gt**, **le**, **ge**
    - More code to maintain
    - Higher chance of inconsistencies
3. When to use @total_ordering:
    - When you need all comparison operators
    - When you want to ensure consistent ordering
    - When you want to minimize code duplication

Iterators and Iterables

```
class NumberSequence:
    def __init__(self, start: int, end: int):
        self.start = start
        self.end = end

    def __iter__(self):
        return NumberIterator(self.start, self.end)

class NumberIterator:
    def __init__(self, start: int, end: int):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
                raise StopIteration
            result = self.current
            self.current += 1
            return result

# Using iterator
numbers = NumberSequence(1, 3)
for num in numbers:
    print(num)  # Prints 1, 2, 3
```

Access Modifiers and Encapsulation

```
class BankAccount:
    def __init__(self, account_number: str, balance: float):
        self.account_number = account_number  # public
        self._balance = balance  # protected
        self.__transactions = []  # private

    def deposit(self, amount: float) -> None:
        if amount > 0:
            self._balance += amount
            self.__transactions.append(('deposit', amount))

    def get_balance(self) -> float:
        return self._balance

    def get_transactions(self) -> list:
        return self.__transactions.copy()

# Using encapsulation
account = BankAccount("123", 1000)
account.deposit(500)
print(account.get_balance())  # 1500
print(account.get_transactions())  # [('deposit', 500)]
```

Key Points:

1. Basic Classes:
    - Constructor (**init**)
    - Instance methods
    - Type annotations
2. Containment:
    - Has-A relationship
    - Using type hints
    - List of objects
3. Inheritance:

- Is-A relationship
- Method overriding
- Parent class methods

4. Abstract Classes:
    - ABC module
    - Abstract methods
    - Implementation requirements
5. Relational Operators:
    - **eq**, **lt**
    - @total_ordering
    - Arithmetic operators
6. Iterators:
    - **iter** and **next**
    - StopIteration
    - For loop support
7. Access Modifiers:
    - Public (no underscore)
    - Protected (_single_underscore)
    - Private (__double_underscore)
    - Encapsulation principles

9. Recursion

Recursion is a programming concept where a function calls itself to solve a problem by breaking it down into smaller subproblems.

Basic Recursion Structure

```python
def recursive_function(n: int) -> int:
    # Base case: stopping condition
    if n <= 0:
        return 0

    # Recursive case: function calls itself
    return n + recursive_function(n - 1)

# Example: Sum of numbers from 1 to n
print(recursive_function(5))  # 15 (5 + 4 + 3 + 2 + 1 + 0)
```

Types of Recursion

1. Direct Recursion

```python
def factorial(n: int) -> int:
    # Base case
    if n <= 1:
        return 1

    # Direct recursive call
    return n * factorial(n - 1)

print(factorial(5))  # 120 (5 * 4 * 3 * 2 * 1)
```

2. Indirect Recursion

```python
def is_even(n: int) -> bool:
    if n == 0:
        return True
    return is_odd(n - 1)

def is_odd(n: int) -> bool:
    if n == 0:
        return False
    return is_even(n - 1)

print(is_even(4))  # True
print(is_odd(4))   # False
```

3. Tail Recursion

```python
def factorial_tail(n: int, accumulator: int = 1) -> int:
    if n <= 1:
        return accumulator
    return factorial_tail(n - 1, n * accumulator)

print(factorial_tail(5))  # 120
```

Common Recursive Problems

1. Fibonacci Sequence

```python
def fibonacci(n: int) -> int:
    # Base cases
    if n <= 0:
        return 0
    if n == 1:
        return 1
    # Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))  # 8 (0, 1, 1, 2, 3, 5, 8)
```

2. Binary Search

```python
def binary_search(arr: list, target: int, left: int = 0, right: int = None) ->
int:
    if right is None:
        right = len(arr) - 1

    # Base case: element not found
    if left > right:
        return -1

    mid = (left + right) // 2

    # Base case: element found
    if arr[mid] == target:
        return mid

    # Recursive cases
    if arr[mid] > target:
        return binary_search(arr, target, left, mid - 1)
    return binary_search(arr, target, mid + 1, right)

arr = [1, 3, 5, 7, 9, 11]
print(binary_search(arr, 7))  # 3
print(binary_search(arr, 6))  # -1
```

3. Greatest Common Divisor (GCD)

```python
def gcd(a: int, b: int) -> int:
    # Base case
    if b == 0:
        return a

    # Recursive case: Euclidean algorithm
    return gcd(b, a % b)

print(gcd(48, 18))  # 6
print(gcd(54, 24))  # 6
```

Recursion with Lists

1. List Sum

```python
def list_sum(lst: list) -> int:
    if not lst:
        return 0
    return lst[0] + list_sum(lst[1:])

print(list_sum([1, 2, 3, 4, 5]))  # 15
```

2. List Reversal

```python
def reverse_list(lst: list) -> list:
    if not lst:
        return []
    return [lst[-1]] + reverse_list(lst[:-1])

print(reverse_list([1, 2, 3, 4]))  # [4, 3, 2, 1]
```

Recursion with Strings

1. String Reversal

```python
def reverse_string(s: str) -> str:
    if not s:
        return ""
    return s[-1] + reverse_string(s[:-1])

print(reverse_string("hello"))  # "olleh"
```

2. Palindrome Check

```python
def is_palindrome(s: str) -> bool:
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])

print(is_palindrome("radar"))  # True
print(is_palindrome("hello"))  # False
```

Key Points:

1. Base Cases:
    - Essential stopping conditions
    - Prevent infinite recursion
    - Usually handle smallest/simplest cases
2. Recursive Cases:
    - Break problem into smaller subproblems
    - Call function with modified parameters
    - Work towards base case
3. Types of Recursion:
    - Direct: Function calls itself
    - Indirect: Functions call each other
    - Tail: Last operation is recursive call
4. Common Applications:
    - String manipulation
    - List operations
    - Basic data processing
    - Simple algorithms
5. Best Practices:
    - Always have base case
    - Ensure progress towards base case
    - Consider stack overflow for deep recursion
    - Use tail recursion when possible
6. Performance Considerations:
    - Space complexity: Stack frames
    - Time complexity: Number of recursive calls
    - Consider iterative solutions for large inputs

10. Sorting and Searching Algorithms

Selection Sort

```python
def selection_sort(arr: list) -> list:
    n = len(arr)
    for i in range(n):
        # Find minimum element in unsorted array
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap found minimum with first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example
arr = [64, 25, 12, 22, 11]
print(selection_sort(arr))  # [11, 12, 22, 25, 64]
```

Time Complexity: O(n^2)

- Outer loop runs n times
- Inner loop runs (n-1) + (n-2) + ... + 1 times
- Total comparisons: n(n-1)/2
- Therefore, O(n^2)

Insertion Sort

```
def insertion_sort(arr: list) -> list:
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements greater than key one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example
arr = [12, 11, 13, 5, 6]
print(insertion_sort(arr))  # [5, 6, 11, 12, 13]
```

Time Complexity: O(n^2)

- Worst case: array is reverse sorted
- Each element needs to be compared with all previous elements
- Total comparisons: 1 + 2 + ... + (n-1) = n(n-1)/2
- Therefore, O(n^2)

Merge Sort

```
def merge_sort(arr: list) -> list:
    if len(arr) <= 1:
        return arr

    # Divide array into two halves
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # Merge the two halves
    return merge(left, right)

def merge(left: list, right: list) -> list:
    result = []
    i = j = 0

    # Compare and merge elements
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Example
arr = [38, 27, 43, 3, 9, 82, 10]
print(merge_sort(arr))  # [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n)

- Each level of recursion divides array in half
- Height of recursion tree: log n
- At each level, merge operation takes O(n)
- Total time: O(n log n)

Quick Sort

```
def quick_sort(arr: list) -> list:
    if len(arr) <= 1:
        return arr

    # Choose pivot (first element)
    pivot = arr[0]

    # Partition array
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]

    # Recursively sort and combine
    return quick_sort(left) + [pivot] + quick_sort(right)
```

```python
# Example
arr = [10, 7, 8, 9, 1, 5]
print(quick_sort(arr))  # [1, 5, 7, 8, 9, 10]
```

Time Complexity:

- Best/Average case: O(n log n)
  - Pivot divides array roughly in half
  - Similar to merge sort's recursion tree
- Worst case: O(n^2)
  - Occurs when array is already sorted
  - Pivot always smallest/largest element
  - Creates unbalanced partitions

Binary Search

```python
def binary_search(arr: list, target: int) -> int:
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # Target not found

# Example
arr = [1, 3, 5, 7, 9, 11]
print(binary_search(arr, 7))  # 3
print(binary_search(arr, 6))  # -1
```

Time Complexity: O(log n)

- Each step reduces search space by half
- Number of steps needed: $\log_2 n$
- Therefore, O(log n)

Bisect Module

```python
import bisect

# bisect_left: Find insertion point to maintain sorted order
arr = [1, 3, 5, 7, 9]
```

```
print(bisect.bisect_left(arr, 6))  # 3

# bisect_right: Find insertion point after any existing entries
print(bisect.bisect_right(arr, 5))  # 3

# insort_left: Insert item in sorted order
bisect.insort_left(arr, 6)
print(arr)  # [1, 3, 5, 6, 7, 9]

# insort_right: Insert item after any existing entries
bisect.insort_right(arr, 5)
print(arr)  # [1, 3, 5, 5, 6, 7, 9]
```

Time Complexity: O(log n)

- Uses binary search internally
- Same complexity as binary search
- Insertion is O(n) due to shifting elements

Key Points:

1. Selection Sort:
    - Simple but inefficient
    - Always O(n^2) regardless of input
    - Good for small arrays
2. Insertion Sort:
    - Efficient for small arrays
    - Adaptive: O(n) for nearly sorted arrays
    - Stable sort
3. Merge Sort:
    - Guaranteed O(n log n)
    - Stable sort
    - Requires extra space
4. Quick Sort:
    - Usually fastest in practice
    - In-place sorting
    - Not stable
5. Binary Search:
    - Requires sorted array
    - Very efficient for large arrays
    - Basis for many algorithms
6. Bisect Module:

  - Built-in binary search
  - Maintains sorted order
  - Useful for insertion points