Module III: Data Structures and Control Flow

1. Introduction to Data Structures

Data structures are fundamental concepts in programming that allow for efficient storage and organization of data. They dictate how data is represented, manipulated, and accessed. Python offers a rich set of built-in data structures that are both powerful and easy to use. This module explores some of the most common and essential data structures, along with Python's control flow mechanisms that dictate the order of execution in a program.

2. Lists

A **list** is one of Python's most versatile mutable sequence types. It is an ordered collection of items, where each item can be of a different data type. Lists are defined using square brackets `[]`, with items separated by commas.

Example:

my_list = [1, "hello", 3.14, True]

List Operations

Lists support various operations:

Concatenation: Joining two lists using the `+` operator.

Repetition: Repeating a list multiple times using the `*` operator.

Membership: Checking if an item exists in a list using the `in` and `not in` operators.

Length: Getting the number of items using the `len()` function.

Example:

list1 = [1, 2, 3]list2 = [4, 5, 6]

combined_list = list1 + list2 # [1, 2, 3, 4, 5, 6]

repeated_list = list1 * 2 # [1, 2, 3, 1, 2, 3]

print(2 in list1) # Trueprint(len(list1)) # 3

List Slicing

Slicing allows you to extract a portion of a list. The syntax is list[start:stop:step] . The start index is inclusive, and the stop index is exclusive. The step indicates the increment.

Example:

numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[2:5]) # [2, 3, 4] (items from index 2 up to, but not including, index 5)

print(numbers[:3]) # [0, 1, 2] (items from the beginning up to index 3)

print(numbers[7:]) # [7, 8, 9] (items from index 7 to the end)

print(numbers[1::2]) # [1, 3, 5, 7, 9] (every second item starting from index 1)

print(numbers[::-1]) # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (reverses the list)

List Methods

Lists have numerous built-in methods to modify or query them:

append(item) : Adds an item to the end of the list.

extend(iterable) : Adds all items from an iterable to the end.

insert(index, item) : Inserts an item at a specific index.

remove(item) : Removes the first occurrence of a specified item.

pop([index]) : Removes and returns the item at a given index (defaults to the last item).

clear() : Removes all items from the list.

index(item) : Returns the index of the first occurrence of an item.

count(item) : Returns the number of occurrences of an item.

sort() : Sorts the list in ascending order.

reverse() : Reverses the order of elements in the list.

Example:

```
fruits = ["apple", "banana"]

fruits.append("cherry") # ["apple", "banana", "cherry"]

fruits.insert(1, "orange") # ["apple", "orange", "banana", "cherry"]

fruits.remove("banana") # ["apple", "orange", "cherry"]

last_fruit = fruits.pop() # last_fruit = "cherry", fruits = ["apple", "orange"]
```

3. Tuples

A **tuple** is similar to a list but is **immutable**, meaning its contents cannot be changed after creation. Tuples are defined using parentheses `()`, with items separated by commas.

Example:

```
my_tuple = (1, "hello", 3.14)
```

Tuples are often used for fixed collections of items, such as coordinates or database records.

Tuple Operations & Slicing

Tuples support many of the same operations and slicing as lists (concatenation, repetition, membership, slicing). However, they do not support in-place modification methods like `append()`, `remove()`, or assignment to elements.

Example:

```
tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

combined_tuple = tuple1 + tuple2 # (1, 2, 3, 4, 5, 6)

print(tuple1[0:2]) # (1, 2)
```

Tuple Methods

Tuples have only two built-in methods:

`count(item)` : Returns the number of occurrences of an item.

index(item) : Returns the index of the first occurrence of an item.

Example:

colors = ("red", "green", "blue", "red")

print(colors.count("red")) # 2

print(colors.index("green")) # 1

4. Sets

A **set** is an unordered collection of unique items. Sets are mutable, but the items within them must be immutable (e.g., numbers, strings, tuples). Sets are defined using curly braces {} or the set() constructor. An empty set must be created using set() , as {} creates an empty dictionary.

Example:

my_set = {1, 2, 3, 3, 2} # {1, 2, 3} (duplicates are automatically removed)

empty_set = set()

Set Operations

Sets are highly optimized for mathematical set operations:

Union (| or union()): Returns a new set with items from both sets.

Intersection (& or intersection()): Returns a new set with items common to both sets.

Difference (- or difference()): Returns a new set with items in the first set but not in the second.

Symmetric Difference (^ or symmetric_difference()): Returns a new set with items in either set, but not both.

Subset (<= or issubset()): Checks if all items of one set are in another.

Superset (>= or issuperset()): Checks if one set contains all items of another.

Example:

```
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}
print(set_a | set_b)        # {1, 2, 3, 4, 5, 6} (Union)
print(set_a & set_b)        # {3, 4} (Intersection)
print(set_a - set_b)        # {1, 2} (Difference)
print(set_a ^ set_b)        # {1, 2, 5, 6} (Symmetric Difference)
```

Set Methods

Sets also have methods for adding and removing elements:

add(item) : Adds a single item to the set.

update(iterable) : Adds all items from an iterable to the set.

remove(item) : Removes an item. Raises KeyError if the item is not found.

discard(item) : Removes an item if present; does nothing if not found.

pop() : Removes and returns an arbitrary element from the set.

clear() : Removes all elements.

Example:

```
letters = {"a", "b"}
letters.add("c") # {"a", "b", "c"}
letters.update(["d", "e"]) # {"a", "b", "c", "d", "e"}
letters.remove("c") # {"a", "b", "d", "e"}
letters.discard("z") # Does nothing, no error
```

5. Dictionaries

A **dictionary** is a mutable, unordered (in Python versions prior to 3.7, ordered from 3.7 onwards) collection of key-value pairs. Each key must be unique and immutable (e.g., strings, numbers, tuples), while values can be of any data

type. Dictionaries are defined using curly braces {} with key-value pairs separated by colons : , and pairs separated by commas.

Example:

student = {"name": "Alice", "age": 20, "major": "Computer Science"}

Dictionary Operations

Key operations involve accessing, adding, modifying, and deleting key-value pairs:

Accessing: Using square brackets dict[key] or the get(key) method.

Adding/Modifying: Using assignment dict[key] = value .

Deleting: Using the del keyword or the pop() method.

Membership: Checking if a key exists using the in operator.

Example:

student = {"name": "Alice", "age": 20}

print(student["name"]) # "Alice"

print(student.get("age")) # 20

student["age"] = 21 # Modifies

agestudent["city"] = "New York" # Adds new key-value pair

del student["major"] # Removes the key-major pair

print("name" in student) # True

Dictionary Methods

Dictionaries have methods to retrieve keys, values, or items:

keys() : Returns a view object displaying a list of all the keys.

values() : Returns a view object displaying a list of all the values.

items() : Returns a view object displaying a list of a dictionary's key-value tuple pairs.

pop(key[, default]) : Removes the specified key and returns the corresponding value.

popitem() : Removes and returns an arbitrary (key, value) pair (last inserted in Python 3.7+).

clear() : Removes all items from the dictionary.

Example:

student = {"name": "Alice", "age": 20}

print(student.keys()) # dict_keys(['name', 'age'])

print(student.values()) # dict_values(['Alice', 20])

print(student.items()) # dict_items([('name', 'Alice'), ('age', 20)])

age = student.pop("age") # age = 20, student = {'name': 'Alice'}

6. Sequences

A sequence is a generic term for an ordered collection of items. In Python, sequences are objects that support indexed access and slicing. Common sequence types include strings, lists, and tuples. They are characterized by their ordered nature and the ability to access elements by their position (index).

Strings: Sequences of characters (immutable).

Lists: Mutable sequences of any data type.

Tuples: Immutable sequences of any data type.

Operations like iteration, concatenation, repetition, and slicing generally apply to all sequence types, though mutability differs.

7. Comprehensions

Comprehensions provide a concise way to create lists, sets, and dictionaries. They are often more readable and efficient than using traditional loops for simple transformations or filtering.

List Comprehensions

The syntax for a list comprehension is [expression for item in iterable if condition] .

Example:

```
# Create a list of squares from 0 to 9
squares = [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# Create a list of even numbers from 0 to 10
even_numbers = [x for x in range(11) if x % 2 == 0] # [0, 2, 4, 6, 8, 10]
```

Dictionary Comprehensions

The syntax for a dictionary comprehension is {key_expression: value_expression for item in iterable if condition} .

Example:

```
# Create a dictionary of squares
square_dict = {x: x**2 for x in range(5)} # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
# Create a dictionary from two lists
keys = ['a', 'b', 'c']values = [1, 2, 3]
dict_from_lists = {k: v for k, v in zip(keys, values)} # {'a': 1, 'b': 2, 'c': 3}
```

8. Introduction to Control Flow

Control flow refers to the order in which statements are executed in a program. Without control flow, programs would simply execute line by line from top to bottom. Python provides constructs like conditional statements and loops to alter this sequential execution, allowing for decision-making and repetition.

9. Conditional Statements

Conditional statements allow a program to execute different blocks of code based on whether certain conditions are true or false.

If Statement

The if statement executes a block of code only if a specified condition is true.

Syntax:

if condition:

code to execute if condition is true

Example:

x = 10

if x > 5:

print("x is greater than 5")

Else Statement

The else statement works with if . The code block under else is executed if the if condition is false.

Syntax:

if condition:

code if true

else:

code if false

Example:

x = 3

if x > 5:

print("x is greater than 5")

else:

print("x is not greater than 5")

Elif Statement

The elif (else if) statement allows you to check multiple conditions in sequence. It is used when you have more than two possible outcomes.

Syntax:

```
if condition1:
# code if condition1 is true
elif condition2:
# code if condition2 is true
else:
# code if all previous conditions are false
```

Example:

```
score = 75
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

10. Loops

Loops are used to execute a block of code repeatedly. Python has two main types of loops: for and while.

For Loop

A for loop iterates over a sequence (like a list, tuple, string, or range) or any other iterable object. It executes the loop body once for each item in the sequence.

Syntax:

for item in iterable:

code to execute for each item

For Loop with Ranges

The range() function is often used with for loops to iterate a specific number of times. range(stop) generates numbers from 0 up to (but not including) stop . range(start, stop) generates from start up to (but not including) stop . range(start, stop, step) includes a step value.

Example:

Iterate 5 times (0 to 4)

for i in range(5):

print(i) # Prints 0, 1, 2, 3, 4# Iterate from 2 to 7 (exclusive)

for j in range(2, 7):

print(j) # Prints 2, 3, 4, 5, 6# Iterate with a step of 2

for k in range(0, 10, 2):

print(k) # Prints 0, 2, 4, 6, 8

For Loop with Strings, Lists, and Dictionaries

Strings: Iterating over a string yields each character.

for char in "Python":

print(char) # P, y, t, h, o, n

Lists: Iterating over a list yields each element.

my_list = [10, 20, 30]for num in my_list:

print(num) # 10, 20, 30

Dictionaries: Iterating directly over a dictionary yields its keys. You can use .values() for values or .items() for key-value pairs.

```
my_dict = {"a": 1, "b": 2}
for key in my_dict: # or my_dict.keys()
print(key) # a, b

for value in my_dict.values():
print(value) # 1, 2

for key, value in my_dict.items():
print(f"Key: {key}, Value: {value}") # Key: a, Value: 1; Key: b, Value: 2
```

While Loop

A while loop executes a block of code as long as a specified condition remains true. It's useful when the number of iterations is not known beforehand.

Syntax:

```
while condition:
# code to execute while condition is true
# IMPORTANT: Ensure the condition eventually becomes false to avoid infinite loops
```

Example:

```
count = 0
while count < 3:
print(f"Count is: {count}")
count += 1 # This updates count, ensuring loop termination
# Prints: Count is: 0, Count is: 1, Count is: 2
```

11. Loop Manipulation

Python provides keywords to control the flow within loops:

Pass Statement

The pass statement is a null operation; nothing happens when it executes. It is used as a placeholder when a statement is syntactically required but no action is needed.

Example:

```
for i in range(5):
if i == 2:
        pass # Do nothing when i is 2 else:
print(i) # Prints 0, 1, 3, 4
```

Continue Statement

The continue statement skips the rest of the current iteration of the loop and proceeds to the next iteration.

Example:

```
for i in range(5):
  if i == 2:
     continue # Skip the rest of the loop body when i is 2
  print(i) # Prints 0, 1, 3, 4
```

Break Statement

The break statement terminates the loop entirely, even if the loop's condition is still true or there are more items in the sequence.

Example:

```
for i in range(5):
  if i == 3:
     break # Exit the loop when i becomes 3
```

```
print(i) # Prints 0, 1, 2
```

Else Clause in Loops

Both for and while loops can have an optional else block. The else block is executed when the loop terminates normally (i.e., the loop condition becomes false for while , or the iterable is exhausted for for), but *not* if the loop is terminated by a break statement.

Example:

```
# For loop with else
for i in range(3):
    print(i)
else:
    print("Loop finished normally.") # Prints 0, 1, 2, then "Loop finished normally."

# For loop with break (else block is skipped)
for i in range(3):
    if i == 1:
        break
    print(i)
else:
    print("This will not be printed.") # Prints 0, then loop breaks. Else is skipped.
```

12. Conditional and Loops Block

Conditional statements (if , elif , else) and loops (for , while) can be nested within each other to create complex program logic. This allows for sophisticated decision-making and repetitive tasks.

Example: Nested loops

Iterating through a list of lists:

```
matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]

for row in matrix:
    print("New row:")
    for element in row:
        print(element)
# Output:
# New row:
# 1
# 2
# 3
# New row:
# 4
# 5
# 6
# New row:
# 7
# 8
# 9
```

Example: Loop within a conditional

Printing even numbers up to a user-defined limit:

```
limit = 10


if limit > 0:
    print(f"Even numbers up to {limit}:")
    for i in range(limit + 1):
        if i % 2 == 0:
            print(i)
# Output:
# Even numbers up to 10:
# 0
# 2
# 4
# 6
# 8
# 10
```