Programming in Python (CSM210-2C)

Unit 2

Data Collections in python

Python offers several built-in data collections that allow you to store, organize, and manipulate data efficiently.

These collections are essential tools for any data-related task.

- **Lists**: Ordered, mutable, allows duplicates.
- **Tuples**: Ordered, immutable, allows duplicates.
- **Sets**: Unordered, mutable, no duplicates.
- **Dictionaries**: Unordered, key-value pairs, keys are unique.

Lists:

What is a List?

A list in Python is an ordered collection of items which can be of different types. Lists are mutable, meaning that you can change their contents (add, remove, or modify elements) after the list has been created.

Creating Lists

You can create a list using square brackets [] or the list() function.

```
# Creating a list using square brackets
my_list = [1, 2, 3, 'hello', 4.5, True]
# Creating a list using the list() function
another_list = list([5, 6, 7, 8])
```

Accessing Elements

You can access elements in a list by their index, with the first element having an index of 0.

```
my_list = [1, 2, 3, 'hello', 4.5, True]
print(my_list[0])  # Output: 1
print(my_list[3])  # Output: 'hello'
```

Negative Indexing

Negative indexing allows you to access elements from the end of the list.

```
print(my_list[-1])  # Output: True
print(my_list[-2])  # Output: 4.5
```

Slicing

Slicing allows you to access a subset of the list.

list[start:end:step]

```
print(my_list[1:4])   # Output: [2, 3, 'hello']
print(my_list[:3])    # Output: [1, 2, 3]
print(my_list[::2])   # Output: [1, 3, 4.5]
print(my_list[::-1])  # Output: [True, 4.5, 'hello', 3, 2, 1]
```

Modifying Lists

Changing Elements

You can change elements by accessing them via their index.

```
my_list[1] = 'changed'
print(my_list)  # Output: [1, 'changed', 3, 'hello', 4.5, True]
```

Adding Elements

You can add elements using the **append(), insert(), and extend()** methods.

append() Method

- **Description**: The append() method adds a single element to the end of the list.
- **Syntax**: list.append(element)
- **Modifies List In-place**: Yes

```
# Create a list
```

```
my_list = [1, 2, 3]
print("Original list:", my_list)  # Output: [1, 2, 3]
# Append an element to the list
my_list.append(4)
print("After append:", my_list)  # Output: [1, 2, 3, 4]
```

extend() Method

- **Description**: The extend() method adds all the elements of an iterable (such as a list, tuple, or string) to the end of the list. It extends the list by appending elements from the iterable.
- **Syntax**: list.extend(iterable)
- **Modifies List In-place**: Yes

```
# Create a list
my_list = [1, 2, 3]
print("Original list:", my_list)  # Output: [1, 2, 3]

# Extend the list with another list
my_list.extend([4, 5, 6])
print("After extend:", my_list)  # Output: [1, 2, 3, 4, 5, 6]
```

insert() Method

- **Description**: The insert() method inserts an element at a specified position in the list.
- **Syntax**: list.insert(index, element)
- **Modifies List In-place**: Yes

```
# Create a list
my_list = [1, 2, 3]
print("Original list:", my_list)  # Output: [1, 2, 3]
```

```
# Insert an element at a specified position
my_list.insert(1, 'inserted')
print("After insert:", my_list)  # Output: [1, 'inserted', 2, 3]
```

Comparison of append(), extend(), and insert()

Method	Adds Element(s)	Location of Addition	Syntax
append()	A single element	End of the list	list.append(element)
extend()	Multiple elements from an iterable	End of the list	list.extend(iterable)
insert()	A single element	Specified position	list.insert(index, element)

Removing Elements

You can remove elements using the remove(), pop(), and clear() methods.

remove() Method

- **Description**: The remove() method removes the first occurrence of a specified value from the list.
- **Syntax**: list.remove(value)
- **Modifies List In-place**: Yes
- **Raises an Error**: Raises a ValueError if the specified value is not found in the list.

```
my_list = [1, 2, 3, 2, 4]
print("Original list:", my_list)  # Output: [1, 2, 3, 2, 4]
```

```
# Remove the first occurrence of the value 2
my_list.remove(2)
print("After remove:", my_list)  # Output: [1, 3, 2, 4]
```

pop() Method

- **Description**: The pop() method removes and returns an element from the list. By default, it removes and returns the last element, but you can specify an index to remove and return an element from a particular position.
- **Syntax**:
  - list.pop(): Removes and returns the last element.
  - list.pop(index): Removes and returns the element at the specified index.
- **Modifies List In-place**: Yes
- **Raises an Error**: Raises an IndexError if the specified index is out of range.

```
my_list = [1, 2, 3, 4]
print("Original list:", my_list)  # Output: [1, 2, 3, 4]

# Pop the last element
popped_element = my_list.pop()
print("After pop:", my_list)       # Output: [1, 2, 3]
print("Popped element:", popped_element)  # Output: 4

# Pop the element at index 1
popped_element = my_list.pop(1)
print("After pop with index:", my_list)  # Output: [1, 3]
print("Popped element:", popped_element)  # Output: 2
```

clear() Method

- **Description**: The clear() method removes all elements from the list, resulting in an empty list.
- **Syntax**: list.clear()
- **Modifies List In-place**: Yes

```
my_list = [1, 2, 3, 4]
print("Original list:", my_list)  # Output: [1, 2, 3, 4]

# Clear all elements from the list
my_list.clear()
print("After clear:", my_list)  # Output: []
```

Comparison of remove(), pop(), and clear()

Method	Description	Syntax	Modifies List In-place	Returns Value	Raises Error
remove()	Removes the first occurrence of a specified value	list.remove(value)	Yes	No	Yes (if value not found)
pop()	Removes and returns the last element or element at a specified index	list.pop([index])	Yes	Yes	Yes (if index out of range)
clear()	Removes all elements from the list	list.clear()	Yes	No	No

List comprehensions

- List comprehensions are a powerful and concise way to create lists in Python.
- They allow for the creation of lists by using a single line of code, often combining loops and conditional statements.

Some Examples

```
squares = [x**2 for x in range(10)]
print(squares)

# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

multiples_of_two = [x * 2 for x in range(10)]
print(multiples_of_two)

# Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

evens = [x for x in range(20) if x % 2 == 0]
print(evens)

# Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

even_squares = [x**2 for x in range(20) if x % 2 == 0]
print(even_squares)

# Output: [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

def square(x):
```

```
    return x**2
numbers = [1, 2, 3, 4, 5]
squared_numbers = [square(x) for x in numbers]
print(squared_numbers)

# Output: [1, 4, 9, 16, 25]

text = "Hello, World!"
vowels = [char for char in text if char.lower() in 'aeiou']
print(vowels)

# Output: ['e', 'o', 'o']

numbers = range(100)
filtered_numbers = [x for x in numbers if x % 2 == 0 and x % 3 == 0]
print(filtered_numbers)

 # Output: [0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]
```

Nested Lists

Lists can contain other lists as elements, creating nested lists.

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list[0][1])

# Output: 2
```

Summary

- **Lists** are versatile and can store elements of any data type.
- They support various operations such as adding, removing, and modifying elements.
- Lists support slicing, allowing access to sub-sections of the list.
- Python provides several built-in methods for lists, which make data manipulation easier.
- List comprehensions offer a concise way to create and manipulate lists.
- Lists can be nested, allowing for complex data structures.

Tuples:

What is a Tuple?

- A tuple is an ordered collection of items which can be of different types.
- Tuples are immutable, meaning that once they are created, their elements cannot be changed, added, or removed.
- This immutability makes tuples useful for protecting data integrity.

Creating Tuples

You can create a tuple by placing a comma-separated sequence of values inside parentheses ().

Parentheses are optional but recommended for readability.

```
# Creating tuples
tuple1 = (1, 2, 3)
tuple2 = ("apple", "banana", "cherry")
tuple3 = (1, "hello", 3.5)

# Creating a tuple without parentheses
tuple4 = 1, 2, 3
# Creating an empty tuple
empty_tuple = ()
# Creating a tuple with a single element (note the trailing comma)
single_element_tuple = (5,)
```

Accessing Tuple Elements

You can access tuple elements by their index, similar to lists. Indexing starts from 0.

```
my_tuple = ("apple", "banana", "cherry")
```

```
# Accessing elements
print(my_tuple[0])  # Output: apple
print(my_tuple[2])  # Output: cherry

# Negative indexing
print(my_tuple[-1])  # Output: cherry
print(my_tuple[-3])  # Output: apple
```

Slicing Tuples

You can extract a subset of a tuple using slicing. The syntax for slicing is tuple[start:end:step].

```
my_tuple = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

# Slicing
print(my_tuple[2:5])  # Output: (2, 3, 4)
print(my_tuple[:3])   # Output: (0, 1, 2)
print(my_tuple[5:])   # Output: (5, 6, 7, 8, 9)
print(my_tuple[::2])  # Output: (0, 2, 4, 6, 8)
print(my_tuple[::-1]) # Output: (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Tuple Immutability

Tuples are immutable, which means you cannot modify, add, or remove elements after the tuple is created. Any attempt to do so will result in an error.

```
my_tuple = (1, 2, 3)
# Attempting to change an element (will raise an error)
# my_tuple[1] = 4  # TypeError: 'tuple' object does not support item assignment
# However, you can reassign the whole tuple
```

```
my_tuple = (4, 5, 6)
print(my_tuple)  # Output: (4, 5, 6)
```

Tuple Operations

Tuples support various operations such as concatenation, repetition, and membership tests.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenation
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)  # Output: (1, 2, 3, 4, 5, 6)

# Repetition
repeated_tuple = tuple1 * 2
print(repeated_tuple)  # Output: (1, 2, 3, 1, 2, 3)

# Membership test
print(2 in tuple1)  # Output: True
print(5 in tuple1)  # Output: False
```

Tuple Methods

Although tuples are immutable, they have two useful methods:

- count(): Returns the number of occurrences of a value.
- index(): Returns the index of the first occurrence of a value.

```
my_tuple = (1, 2, 3, 2, 2, 4, 5)
```

```
# count()
print(my_tuple.count(2))  # Output: 3

# index()
print(my_tuple.index(2))  # Output: 1
```

Tuple Unpacking

Tuple unpacking allows you to assign the elements of a tuple to multiple variables in a single statement.

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

# Unpacking
a, b, c = my_tuple
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Nested Tuples

Tuples can contain other tuples, creating nested tuples.

```
nested_tuple = (1, (2, 3), (4, (5, 6)))
print(nested_tuple)  # Output: (1, (2, 3), (4, (5, 6)))

# Accessing nested tuple elements
print(nested_tuple[1])    # Output: (2, 3)
print(nested_tuple[2][1]) # Output: (5, 6)
```

Advantages of Tuples

- **Immutability**: Once created, tuples cannot be changed, which ensures the integrity of the data.
- **Faster**: Tuples are generally faster than lists due to their immutability.
- **Memory Efficient**: Tuples use less memory than lists.
- **Hashable**: Tuples can be used as keys in dictionaries if they contain only hashable types.
-

Use Cases of Tuples

- **Return Multiple Values**: Functions can return multiple values using tuples.
- **Unchanging Data**: Use tuples for data that should not change throughout the program.
- **Dictionary Keys**: Use tuples as keys in dictionaries for complex key structures.
-

Summary

- Tuples are ordered and immutable collections of items.
- They support indexing, slicing, and various operations like concatenation and repetition.
- Tuples can be nested and can be unpacked into variables.
- Tuples have some performance advantages over lists due to their immutability.

Dictionaries in Python

Dictionaries are a fundamental data structure in Python that store data in key-value pairs.

Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be of any immutable type.

Key Features of Dictionaries

1. **Unordered**: Dictionaries do not maintain any order for their elements.
2. **Mutable**: You can change, add, or remove items after the dictionary is created.
3. **Indexed by Keys**: Each value in the dictionary is associated with a unique key, and you access the value by using the key.
4. **Dynamic Size**: Dictionaries can grow and shrink as needed.

Creating Dictionaries

You can create dictionaries using curly braces {} with key-value pairs separated by colons, or by using the dict() constructor.

```
# Using curly braces
student = {
    'name': 'Alice',
    'age': 21,
    'courses': ['Math', 'Science']
}
# Using dict() constructor
student = dict(name='Alice', age=21, courses=['Math', 'Science'])
```

Accessing Values

You can access values in a dictionary by using the keys.

```
student = {
    'name': 'Alice',
    'age': 21,
    'courses': ['Math', 'Science']
}

# Accessing values
print(student['name'])  # Output: Alice
print(student['age'])   # Output: 21
```

 # Using get() method

The get() method in Python dictionaries is a very useful tool for safely accessing the values associated with specific keys.

It allows you to retrieve the value for a given key if the key exists in the dictionary, and return a default value if the key is not found.

This helps in avoiding KeyError exceptions when a key is missing.

dictionary.get(key, default_value)

- key: The key you want to look up in the dictionary.
- default_value: (Optional) The value to return if the key is not found. If omitted, None is returned by default.

```
print(student.get('courses'))  # Output: ['Math', 'Science']
print(student.get('grade', 'Not Available'))  # Output: Not Available
```

Modifying Dictionaries

You can add, update, or remove key-value pairs in a dictionary.

```
student = { 'name': 'Alice',     'age': 21,     'courses': ['Math', 'Science'] }

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

# Updating an existing key-value pair
student['age'] = 22

# Removing a key-value pair using del
del student['courses']

# Removing a key-value pair using pop()
age = student.pop('age')

print(student)  # Output: {'name': 'Alice', 'grade': 'A'}
print('Removed age:', age)  # Output: Removed age: 22
```

Looping Through Dictionaries

You can iterate through the keys, values, or key-value pairs of a dictionary.

```
student = {  'name': 'Alice',     'age': 21,     'grade': 'A'       }

# Looping through keys
for key in student:
    print(key)
# Looping through values
for value in student.values():
    print(value)
```

 # Looping through key-value pairs

```
for key, value in student.items():
    print(f'{key}: {value}')
```

Dictionary methods

1. clear() : Removes all items from the dictionary.

```
student = {'name': 'Alice', 'age': 21, 'grade': 'A'}
student.clear()
print(student)  # Output: {}
```

2. copy() : Returns a shallow copy of the dictionary.

```
student = {'name': 'Alice', 'age': 21, 'grade': 'A'}
student_copy = student.copy()
print(student_copy)  # Output: {'name': 'Alice', 'age': 21, 'grade': 'A'}
```

3. fromkeys(seq[, value]) : Creates a new dictionary with keys from seq and values set to value.

```
keys = ['name', 'age', 'grade']
student = dict.fromkeys(keys, None)
print(student)  # Output: {'name': None, 'age': None, 'grade': None}
```

4. get(key[, default])

Returns the value for key if key is in the dictionary; otherwise returns default.

```
student = {'name': 'Alice', 'age': 21}
age = student.get('age', 'Not Available')
```

```
grade = student.get('grade', 'Not Available')
print(age)  # Output: 21
print(grade)  # Output: Not Available
```

5. items()

Returns a view object that displays a list of a dictionary's key-value tuple pairs.

```
student = {'name': 'Alice', 'age': 21}
items = student.items()
print(items)  # Output: dict_items([('name', 'Alice'), ('age', 21)])
```

6. keys()

Returns a view object that displays a list of all the keys in the dictionary.

```
student = {'name': 'Alice', 'age': 21}
keys = student.keys()
print(keys)  # Output: dict_keys(['name', 'age'])
```

7. pop(key[, default])

Removes the item with the specified key and returns its value. If the key is not found, default is returned if provided; otherwise, it raises a KeyError.

```
student = {'name': 'Alice', 'age': 21}
age = student.pop('age')
print(age)  # Output: 21
print(student)  # Output: {'name': 'Alice'}

# Using default value
grade = student.pop('grade', 'Not Available')
print(grade)  # Output: Not Available
```

popitem()

Removes and returns the last inserted key-value pair as a tuple. This method is useful in Python 3.7+ where dictionaries maintain insertion order.

```
student = {'name': 'Alice', 'age': 21}
item = student.popitem()
print(item)  # Output: ('age', 21)
print(student)  # Output: {'name': 'Alice'}
```

9. setdefault(key[, default])

Returns the value of the key if it is in the dictionary. If not, inserts key with a value of default and returns default.

```
student = {'name': 'Alice'}
age = student.setdefault('age', 21)
grade = student.setdefault('grade', 'A')
print(student)  # Output: {'name': 'Alice', 'age': 21, 'grade': 'A'}
print(age)  # Output: 21
print(grade)  # Output: A
```

10. update([other])

Updates the dictionary with the key-value pairs from another dictionary or from an iterable of key-value pairs.

```
student = {'name': 'Alice', 'age': 21}
update_info = {'grade': 'A', 'age': 22}
student.update(update_info)
print(student)  # Output: {'name': 'Alice', 'age': 22, 'grade': 'A'}
```

```
# Using an iterable of key-value pairs
student.update(name='Bob', courses=['Math', 'Science'])
print(student)  # Output: {'name': 'Bob', 'age': 22, 'grade': 'A', 'courses': ['Math', 'Science']}
```

11. values()

Returns a view object that displays a list of all the values in the dictionary.

```
student = {'name': 'Alice', 'age': 21}
values = student.values()
print(values)                # Output: dict_values(['Alice', 21])
```

Example of Dictionary Comprehension

Like list comprehensions, you can create dictionaries using dictionary comprehensions for more concise and readable code.

```
# Creating a dictionary of squares
squares = {x: x**2 for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Practical Applications of Dictionaries

- **Storing Configuration Settings**: Easily store and access configuration parameters using keys.
- **Counting Items**: Count occurrences of items in a list.
- **Storing Objects with Unique Identifiers**: Store objects using unique keys, such as IDs or names.

Summary

- Dictionaries are an essential data structure in Python for storing key-value pairs.
- They are unordered, mutable, and indexed by unique keys.
- You can create, access, modify, and loop through dictionaries using various methods and techniques.
- Dictionary comprehensions provide a concise way to create dictionaries.
- Practical applications of dictionaries include storing configuration settings, counting items, and more.

Sorted() function in python

The sorted() function in Python is a built-in function that returns a new sorted list from the elements of any iterable.

It can be used to sort lists, tuples, strings, and more.

Unlike the sort() method, which modifies the list in place, sorted() creates a new list and leaves the original iterable unchanged.

sorted(iterable, key=None, reverse=False)

- iterable: The sequence (e.g., list, tuple, string) to be sorted.
- key (optional): A function to be called on each element prior to making comparisons (e.g., str.lower for case-insensitive sorting).
- reverse (optional): If True, the list elements are sorted as if each comparison were reversed.

Sorting a list of numbers in ascending order:

```
numbers = [3, 1, 4, 1, 5, 9, 2]

sorted_numbers = sorted(numbers)

print(sorted_numbers)

# Output: [1, 1, 2, 3, 4, 5, 9]
```

Sorting in Descending Order

Using the reverse parameter to sort in descending order:

```
numbers = [3, 1, 4, 1, 5, 9, 2]

sorted_numbers = sorted(numbers, reverse=True)

print(sorted_numbers)

# Output: [9, 5, 4, 3, 2, 1, 1]
```

Sorting with a Key Function

Sorting a list of strings case-insensitively:

```
words = ['banana', 'Apple', 'cherry', 'date']
sorted_words = sorted(words, key=str.lower)
print(sorted_words)
# Output: ['Apple', 'banana', 'cherry', 'date']
```

Sorting a List of Tuples

Sorting a list of tuples based on the second element:

```
students = [('Alice', 25), ('Bob', 20), ('Charlie', 23)]
sorted_students = sorted(students, key=lambda x: x[1])
print(sorted_students)  # Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]
```

Sorting a Dictionary by Keys

Sorting the keys of a dictionary:

```
grades = {'Alice': 85, 'Bob': 92, 'Charlie': 87}
sorted_keys = sorted(grades)
print(sorted_keys)  # Output: ['Alice', 'Bob', 'Charlie']
```

Sorting a Dictionary by Values

Sorting the items of a dictionary by values:

```
grades = {'Alice': 85, 'Bob': 92, 'Charlie': 87}
sorted_grades = sorted(grades.items(), key=lambda x: x[1])
print(sorted_grades)  # Output: [('Alice', 85), ('Charlie', 87), ('Bob', 92)]
```

Sorting a List of Dictionaries

Sorting a list of dictionaries by a specific key:

```
students = [
    {'name': 'Alice', 'grade': 'A'},
    {'name': 'Bob', 'grade': 'C'},
    {'name': 'Charlie', 'grade': 'B'}
]
sorted_students = sorted(students, key=lambda x: x['grade'])
print(sorted_students)
# Output: [{'name': 'Alice', 'grade': 'A'}, {'name': 'Charlie', 'grade': 'B'}, {'name': 'Bob', 'grade': 'C'}]
```

Zip()

The zip() function in Python is a powerful tool for working with multiple iterables in parallel.

It takes two or more iterables (like lists, tuples, or strings) and aggregates them into an iterator of tuples.

Each tuple contains elements from the iterables at corresponding positions.

How zip() Works

Aggregating Elements:

- zip() takes iterables as arguments and returns an iterator of tuples.
- Each tuple contains one element from each of the input iterables, taken at the same index.

  Zipping Two Lists

  ```
  # Two lists of equal length
  list1 = [1, 2, 3]
  list2 = ['a', 'b', 'c']
  # Using zip()
  zipped = zip(list1, list2)
  print(zipped)  ---------------------→
  # Converting the iterator to a list
  zipped_list = list(zipped)
  print(zipped_list)
  # Output: [(1, 'a'), (2, 'b'), (3, 'c')]
  ```

  Zipping Lists of Different Lengths

```
# Lists of different lengths
list1 = [1, 2, 3, 4]
list2 = ['a', 'b']
# Using zip()
zipped = zip(list1, list2)
# Converting the iterator to a list
zipped_list = list(zipped)
print(zipped_list)  # Output: [(1, 'a'), (2, 'b')]
```

Unzipping Using zip()

You can "unzip" a list of tuples into separate lists using the unpacking operator *.

```
# List of tuples
zipped_list = [(1, 'a'), (2, 'b'), (3, 'c')]
# Unzipping
list1, list2 = zip(*zipped_list)
print(list(list1))  # Output: [1, 2, 3]
print(list(list2))  # Output: ['a', 'b', 'c']
```

Here are some common uses of the zip() function:

1. Pairing Elements from Multiple Iterables

One of the most common uses of zip() is to pair elements from two or more iterables.

This is particularly useful when you have related data in separate lists and you want to process them together.

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 92, 78]
paired = zip(names, scores)
for name, score in paired:
```

```
    print(f'{name}: {score}')
# Output:
# Alice: 85
# Bob: 92
# Charlie: 78
```

Creating Dictionaries

You can use zip() to create dictionaries by combining two lists, one with keys and one with values.

Parallel Iteration

When you need to iterate over multiple iterables at the same time, zip() allows you to do so efficiently.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
for a, b in zip(list1, list2):
    print(a + b)
# Output:
# 5
# 7
# 9
```

Combining Iterables of Different Lengths

zip() stops when the shortest iterable is exhausted, making it useful when you want to pair elements but handle differing lengths gracefully.

Is Zip() function is Lazy ?

Yes, zip() is lazy in Python. It returns an iterator that generates tuples only as needed, rather than creating the entire list of tuples at once. This behaviour is more memory efficient, especially when dealing with large datasets.

Lambda Functions in Python

- A **lambda function** in Python is a small, anonymous function defined with the lambda keyword.
- Lambda functions can have any number of arguments but only one expression.
- They are often used for short, simple functions that are not complex enough to require a full function definition using the def keyword.

Basic Syntax

The syntax for a lambda function is:

lambda arguments: expression

A basic lambda function that adds two numbers:

```
add = lambda x, y  :   x + y

print(add(2, 3))

# Output: 5
```

- lambda x, y: defines the arguments.
- x + y is the expression that gets evaluated and returned.

Lambda for Conditional Expressions

Using a lambda function to implement a conditional expression.

```
max_value = lambda a, b: a if a > b else b

print(max_value(10, 15))

# Output: 15
```

Lambda with List Comprehension

Using a lambda function within a list comprehension to apply a transformation.

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = [(lambda x: x ** 2)(x) for x in numbers]

print(squared_numbers)

 Output: [1, 4, 9, 16, 25]
```

Characteristics of Lambda Functions

1. **Anonymous**: Lambda functions are nameless; they are defined without a name.
2. **Single Expression**: They can only contain a single expression (no statements or complex logic).
3. **Inline**: Often used inline within other functions or methods.

When to Use Lambda Functions

1. **Small, Throwaway Functions**: For simple tasks that are used once or a few times.
2. **Arguments to Higher-Order Functions**: Functions that take other functions as arguments, like map(), filter(), and reduce()

map():

The map() function in Python is a built-in function used to apply a given function to all items in an iterable (such as a list or tuple) and return a map object (an iterator) of the results. It's a powerful tool for transforming data in a concise and readable way.

Syntax

map(function, iterable, ...)

- **function: The function to apply to each item.**
- **iterable: One or more iterables (such as lists, tuples, or strings).**

Example: Using a Named Function with map()

```
def square(x):
    return x ** 2
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)

# Convert the map object to a list to see the results
```

```
print(list(squared_numbers))
# Output: [1, 4, 9, 16, 25]
```

Example: Using a Named Function with Lambda()

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x ** 2, numbers)

# Convert the map object to a list to see the results
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

Example: Mapping Over Multiple Iterables

You can use map() with multiple iterables.

In this case, the function should take as many arguments as there are iterables.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
sum_lists = map(lambda x, y: x + y, list1, list2)

# Convert the map object to a list to see the results
print(list(sum_lists))  # Output: [5, 7, 9]

words = ['hello', 'world']
upper_words = map(str.upper, words)
# Convert the map object to a list to see the results
print(list(upper_words))  # Output: ['HELLO', 'WORLD']
```

What is reduce()?

The reduce() function in Python is used to apply a function cumulatively to the items of an iterable, like a list, from left to right, so as to reduce the iterable to a single cumulative value.

Step-by-Step Example: Summing Elements in a List

Let's use reduce() to sum the elements of a list [1, 2, 3, 4, 5].

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

sum_numbers = reduce(lambda x, y: x + y, numbers)

print(sum_numbers)  # Output: 15
```

Step-by-Step Breakdown:

1. **Initial State**:
    - The list is [1, 2, 3, 4, 5].
    - The lambda function is lambda x, y: x + y.
2. **First Iteration**:
    - x = 1 (first element)
    - y = 2 (second element)
    - Result of lambda x, y: x + y is 1 + 2 = 3.
3. **Second Iteration**:
    - x = 3 (result from the first iteration)
    - y = 3 (third element)
    - Result of lambda x, y: x + y is 3 + 3 = 6.
4. **Third Iteration**:
    - x = 6 (result from the second iteration)
    - y = 4 (fourth element)
    - Result of lambda x, y: x + y is 6 + 4 = 10.

5. **Fourth Iteration**:
    - x = 10 (result from the third iteration)
    - y = 5 (fifth element)
    - Result of lambda x, y: x + y is 10 + 5 = 15.
6. **Final Result**:
    - The cumulative value is 15.

Example with Initializer

When you provide an initializer, it acts as the starting value for the reduction.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_numbers = reduce(lambda x, y: x + y, numbers, 10)
print(sum_numbers)  # Output: 25
```

Step-by-Step Breakdown with Initializer:

1. **Initial State**:
    - The list is [1, 2, 3, 4, 5].
    - The lambda function is lambda x, y: x + y.
    - Initializer is 10.
2. **First Iteration**:
    - x = 10 (initializer)
    - y = 1 (first element)
    - Result of lambda x, y: x + y is 10 + 1 = 11.
3. **Second Iteration**:
    - x = 11 (result from the first iteration)

- y = 2 (second element)
- Result of lambda x, y: x + y is 11 + 2 = 13.

4. **Third Iteration**:
    - x = 13 (result from the second iteration)
    - y = 3 (third element)
    - Result of lambda x, y: x + y is 13 + 3 = 16.
5. **Fourth Iteration**:
    - x = 16 (result from the third iteration)
    - y = 4 (fourth element)
    - Result of lambda x, y: x + y is 16 + 4 = 20.
6. **Fifth Iteration**:
    - x = 20 (result from the fourth iteration)
    - y = 5 (fifth element)
    - Result of lambda x, y: x + y is 20 + 5 = 25.
7. **Final Result**:
    - The cumulative value is 25.

Key Points of Reduce function:

- **Binary Function**: The function you pass to reduce() must take exactly two arguments.
- **Cumulative Operation**: The function is applied cumulatively to pairs of items in the iterable.
- **Optional Initializer**: If provided, the initializer is used as the starting value.

Exercises on Reduce function

Product of List Elements: Use the reduce() function to calculate the product of all elements in a list.

numbers = [1, 2, 3, 4, 5]

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product_numbers = reduce(lambda x, y: x * y, numbers)
print(product_numbers)  # Output: 120
```

Find Maximum Element: Use the reduce() function to find the maximum element in a list.

numbers = [1, 2, 3, 4, 5]

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
max_number = reduce(lambda x, y: x if x > y else y, numbers)
print(max_number)  # Output: 5
```

Concatenate Strings: Use the reduce() function to concatenate all strings in a list into a single string.

words = ["Hello", "world", "from", "Python"]

```
from functools import reduce
words = ["Hello", "world", "from", "Python"]
sentence = reduce(lambda x, y: x + " " + y, words)
print(sentence)  # Output: Hello world from Python
```

Factorial Calculation : Use the reduce() function to calculate the factorial of a number.

The factorial of n is the product of all positive integers less than or equal to n.

```
from functools import reduce

n = 5

factorial = reduce(lambda x, y: x * y, range(1, n + 1))

print(factorial)  # Output: 120
```

Flatten a List of Lists: Use the reduce() function to flatten a list of lists into a single list.

```
from functools import reduce

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

flattened_list = reduce(lambda x, y: x + y, lists)

print(flattened_list)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Calculate the GCD: Use the reduce() function to find the Greatest Common Divisor (GCD) of a list of numbers.

```
from functools import reduce

import math

numbers = [48, 64, 16]

gcd = reduce(math.gcd, numbers)

print(gcd)  # Output: 16
```

MCQ

1. **What is the correct syntax for a lambda function that adds two numbers?**
   a. a) lambda x, y: x + y b) def lambda x, y: x + y
   b. c) lambda: x, y -> x + y d) lambda (x, y) => x + y
   c. **Answer:** a) lambda x, y: x + y
2. **What does the** map() **function return?**
   a. a) A list b) A tuple c) A map object d) A dictionary
   b. **Answer:** c) A map object
3. **What is the output of the following code?**

   ```
   numbers = [1, 2, 3, 4]

   squared = map(lambda x: x ** 2, numbers)

   print(list(squared))
   ```

   a) [1, 2, 3, 4] b) [1, 4, 9, 16] c) [2, 4, 6, 8] d) [1, 8, 27, 64]
   Answer: b) [1, 4, 9, 16]
4. **Which module do you need to import to use the** reduce() **function?**
   a) itertools b) functools c) collections d) operator
   Answer: b) functools
5. What is the output of the following code?

   ```
   from functools import reduce

   numbers = [1, 2, 3, 4]

   product = reduce(lambda x, y: x * y, numbers)

   print(product)
   ```

   a) 10 b) 24 c) 120 d) 15

   Answer: b) 24

filter() in python

The filter() function in Python is a powerful tool for constructing an iterator from elements of an iterable (like lists, tuples, etc.) that satisfy a function.

This function is often used for filtering data based on specific conditions.

Syntax

filter(function, iterable)

function: A function that tests each element of the iterable and returns True or False.

iterable: An iterable like a list, tuple, etc.

Basic Functionality

The filter() function applies the specified function to each element in the iterable.

If the function returns True, the element is included in the returned filter object; if False, it is excluded.

Example: Filtering Even Numbers

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = filter(lambda x: x % 2 == 0, numbers)


print(list(even_numbers))

# Output: [2, 4, 6, 8, 10]
```

Example: Filtering Strings

Filtering a list of strings to include only those with more than 5 characters.

```
words = ["apple", "banana", "cherry", "date", "fig", "grape"]

long_words = filter(lambda x: len(x) > 5, words)

print(list(long_words))

# Output: ['banana', 'cherry']
```

Example: Using a Named Function with filter()

Using a named function to filter elements.

```
def is_positive(num):
    return num > 0

numbers = [-10, -5, 0, 5, 10]
positive_numbers = filter(is_positive, numbers)
print(list(positive_numbers))  # Output: [5, 10]
```

Example: Filtering Dictionaries

Filtering a list of dictionaries to include only those where the value of a specific key meets a condition.

```
students = [
        {'name': 'Alice', 'grade': 'A'},
        {'name': 'Bob', 'grade': 'C'},
        {'name': 'Charlie', 'grade': 'B'}
    ]

# Filter students with grade 'A'
grade_a_students = filter(lambda x: x['grade'] == 'A', students)
print(list(grade_a_students))  # Output: [{'name': 'Alice', 'grade': 'A'}]
```

Example: Filtering Characters in a String

Filtering out vowels from a string.

```
    text = "Hello World"
    vowels = "aeiouAEIOU"
    filtered_text = filter(lambda x: x not in vowels, text)
    print(''.join(filtered_text))  # Output: "Hll Wrld"
```

Example: Filtering with Multiple Conditions

Filtering a list of numbers to include only those that are both even and greater than 5.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered_numbers = filter(lambda x: x % 2 == 0 and x > 5, numbers)

print(list(filtered_numbers))  # Output: [6, 8, 10]
```

Understanding the Behavior of filter()

- **Returns an Iterator**: The filter() function returns an iterator, which is a more memory-efficient way to handle large datasets because it doesn't create a full list in memory.
- **Lazy Evaluation**: The elements are evaluated lazily, meaning they are processed only when you iterate over the filter object. To see the results immediately, you often convert the filter object to a list using list().

Sets in Python

A Set in Python is used to store a collection of items with the following properties.

- No duplicate elements. If try to insert the same item again, it overwrites previous one.
- An unordered collection. When we access all items, they are accessed without any specific order and we cannot access items using indexes as we do in lists.
- Internally use hashing that makes set efficient for search, insert and delete operations. It gives a major advantage over a list for problems with these operations.
- Mutable, meaning we can add or remove elements after their creation, the individual elements within the set cannot be changed directly.

Sets are unordered collections of unique elements in Python.

They are useful when you need to store non-duplicate items and perform common set operations like unions and intersections.

Python sets are mutable, meaning their contents can be changed after creation.

However, Python also provides a variant called frozenset, which is immutable.

Creating Sets

You can create a set using curly braces {} or the set() function.

```
# Creating a set with curly braces
fruits = {'apple', 'banana', 'cherry'}
print(fruits)  # Output: {'cherry', 'banana', 'apple'}
```

```
# Creating a set with the set() function
numbers = set([1, 2, 3, 4, 5])
print(numbers)  # Output: {1, 2, 3, 4, 5}
```

Iterating Over Sets

You can iterate over a set using a for loop.

```
fruits = {'apple', 'banana', 'cherry'}
for fruit in fruits:
    print(fruit)
```

Set Methods

Python sets come with several built-in methods for common set operations:

1. **add()**: Adds an element to the set.

   ```
   fruits.add('orange')
   print(fruits)
   # Output: {'cherry', 'banana', 'apple', 'orange'}
   ```

2. **remove()**: Removes an element from the set. Raises a KeyError if the element is not present.

   ```
   fruits.remove('banana')
   print(fruits)              # Output: {'cherry', 'apple', 'orange'}
   ```

3. **discard()**: Removes an element from the set if it is present. Does nothing if the element is not present.

   ```
   fruits.discard('banana')
   print(fruits)  # Output: {'cherry', 'apple', 'orange'}
   ```

4. **pop()**: Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

   ```
   fruit = fruits.pop()
   print(fruit)
   print(fruits)
   ```

5. **clear()**: Removes all elements from the set.

   ```
   fruits.clear()
   print(fruits)  # Output: set()
   ```

6. **copy()**: Returns a shallow copy of the set.

   ```
   new_fruits = fruits.copy()
   ```

Set Operators

Python provides several operators for set operations:

1. **Union (|)**: Returns a set containing all elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

2. **Intersection (&)**: Returns a set containing elements that are in both sets.

   ```
   intersection_set = set1 & set2
   print(intersection_set)  # Output: {3}
   ```

3. **Difference (-)**: Returns a set containing elements that are in the first set but not in the second set.

   ```
   difference_set = set1 - set2
   print(difference_set)  # Output: {1, 2}
   ```

4. **Symmetric Difference (^)**: Returns a set containing elements that are in either set, but not in both.

   ```
   sym_diff_set = set1 ^ set2
   print(sym_diff_set)  # Output: {1, 2, 4, 5}
   ```

Frozen Sets

A frozenset is an immutable version of a set. Once created, its contents cannot be changed.

Example:

```
# Creating a frozenset
immutable_set = frozenset([1, 2, 3, 4])
print(immutable_set)

# Attempting to modify a frozenset will raise an error
# immutable_set.add(5)  # Raises AttributeError
```

Summary

Sets in Python are a versatile and powerful tool for managing collections of unique elements.

They offer a wide range of methods and operators for performing set operations.

Additionally, frozenset provides an immutable alternative for situations where a constant set is required.