**Module 10: Collections**

* **namedtuple()**

In Python, namedtuple() is a function from the collections module that lets you create tuple-like objects with named fields. Unlike regular tuples, where you access elements using indices (e.g., t[0]), with namedtuple, you can use named attributes (e.g., t.name), which improves code readability.

**✅ Syntax:**

from collections import namedtuple

NamedTupleName = namedtuple('NamedTupleName', ['field1', 'field2', 'field3', ...])

Example:

from collections import namedtuple

# Define a namedtuple called 'Student'

Student = namedtuple('Student', ['name', 'age', 'grade'])

# Create an instance of Student

s1 = Student(name='Amit', age=21, grade='A')

# Access values

print(s1.name) # Output: Amit

print(s1.age) # Output: 21

print(s1.grade) # Output: A

# Access using index (like regular tuple)

print(s1[0]) # Output: Amit

Real-World Example

Suppose you want to store **customer account details**:  
Each account has the following fields:

* account\_number
* name
* balance

from collections import namedtuple

# Define the namedtuple

BankAccount = namedtuple('BankAccount', ['account\_number', 'name', 'balance'])

# Create a few bank accounts

account1 = BankAccount('1234567890', 'Amit Sharma', 25000.50)

account2 = BankAccount('9876543210', 'Priya Mehta', 17500.00)

# Access data using dot notation

print(f"Account Holder: {account1.name}")

print(f"Account Number: {account1.account\_number}")

print(f"Balance: ₹{account1.balance}")

# Let's list all accounts

accounts = [account1, account2]

# Display all accounts

print("\n--- All Bank Accounts ---")

for acc in accounts:

print(f"{acc.name} ({acc.account\_number}) - Balance: ₹{acc.balance}")

* **deque**

deque stands for double-ended queue.  
It allows you to add and remove elements from both ends (left and right) very efficiently — much faster than using lists for these operations.

Example:

from collections import deque

# Create a deque

dq = deque([1, 2, 3])

# Add to the right

dq.append(4)

print(dq)

# Add to the left

dq.appendleft(0)

print(dq)

# Remove from right

dq.pop()

print(dq)

# Remove from left

dq.popleft()

print(dq)  # Output: deque([1, 2, 3])

Real-World Example:

from collections import deque

# Simulate a task queue

task\_queue = deque()

# Add tasks

task\_queue.append("Print Invoice")

task\_queue.append("Send Email")

task\_queue.append("Backup Files")

# Process tasks (FIFO)

print("Processing tasks:")

while task\_queue:

    task = task\_queue.popleft()

    print(f"Task completed: {task}")

* ChainMap

ChainMap is a class in the collections module that **groups multiple dictionaries (or mappings) together into a single view**. This is useful when you want to search through multiple dictionaries as if they were one.

Syntax:

from collections import ChainMap

cm = ChainMap(dict1, dict2, ...)

Example:

from collections import ChainMap

default\_settings = {

'font': 'Arial',

'theme': 'light',

'autosave': True,

}

user\_settings = {

'theme': 'dark'

}

# Chain user settings first to override defaults

settings = ChainMap(user\_settings, default\_settings)

print(settings['font']) # Output: Arial (from default)

print(settings['theme']) # Output: dark (from user override)

print(settings['autosave']) # Output: True (from default)

* **Counter**

Counter is a **dictionary subclass** used for **counting hashable objects**. It’s part of the collections module and is very useful when you want to count the frequency of elements in a list, string, or any iterable.

Syntax:

from collections import Counter

Counter(iterable) # like a list or string

Example 1: Count elements in a list

from collections import Counter

fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']

count = Counter(fruits)

print(count)

# Output: Counter({'apple': 3, 'banana': 2, 'orange': 1})

Example 2: Count characters in a string

from collections import Counter

word = "programming"

char\_count = Counter(word)

print(char\_count)

# Output: Counter({'g': 2, 'r': 2, 'o': 1, 'p': 1, 'a': 1, 'm': 2, 'i': 1, 'n': 1})

Example 3: Using .most\_common()

from collections import Counter

nums = [1, 2, 2, 3, 3, 3, 4]

c = Counter(nums)

print(c.most\_common(2)) # [(3, 3), (2, 2)]

Real-Life Example: Word Frequency in a Sentence

from collections import Counter

sentence = "the quick brown fox jumps over the lazy dog the quick fox"

words = sentence.split()

count = Counter(words)

print(count)

# Output: Counter({'the': 3, 'quick': 2, 'fox': 2, 'brown': 1, ...})

* **OrderedDict**

OrderedDict is a **dictionary subclass** from the collections module that **remembers the order in which items are inserted**.

Example 1: Basic Usage

from collections import OrderedDict

od = OrderedDict()

od['apple'] = 3

od['banana'] = 1

od['orange'] = 2

print(od)

Example : Real-Life Use – Maintaining a Sequence of Tasks

tasks = OrderedDict()

tasks['task1'] = 'Open email'

tasks['task2'] = 'Reply to messages'

tasks['task3'] = 'Attend meeting'

for task, action in tasks.items():

print(task, ":", action)

* **defaultdict**

defaultdict is a **dictionary subclass** that automatically provides a **default value** for missing keys, so you don’t get a KeyError when accessing them.

Example:

**Organizing students by their classes**

We have a list of (class, student) pairs and we want to group students automatically without writing extra checks.

from collections import defaultdict

# Class → Students mapping

class\_students = defaultdict(list)

# Data from school records

data = [

('Class 1', 'Amit'),

('Class 2', 'Priya'),

('Class 1', 'Rohan'),

('Class 3', 'Sneha'),

('Class 2', 'Vikas'),

('Class 1', 'Meera')

]

# Group students by class

for class\_name, student in data:

class\_students[class\_name].append(student)

# Display results

for class\_name, students in class\_students.items():

print(f"{class\_name}: {', '.join(students)}")

* **UserDict**

UserDict is a **wrapper around the built-in dictionary** that makes it easier to create custom dictionary-like classes by **inheriting** from it instead of directly inheriting from dict.

Example: **Scenario:** Phonebook that prevents empty phone numbers

We want a phonebook dictionary, but we don’t want anyone to save a contact without a valid number.

from collections import UserDict

class PhoneBook(UserDict):

def \_\_setitem\_\_(self, name, phone):

if not phone or not phone.strip():

raise ValueError("Phone number cannot be empty!")

super().\_\_setitem\_\_(name, phone)

# Using the custom PhoneBook

pb = PhoneBook()

pb['Amit'] = '9876543210' # ✅ Works

pb['Priya'] = '9123456789' # ✅ Works

try:

pb['Rohan'] = '' # ❌ Empty number

except ValueError as e:

print(e)

print(pb)

* **UserList**

UserList is a **wrapper around the built-in list**, just like UserDict is for dictionaries.  
It lets you create **custom list-like classes** without worrying about tricky behavior when subclassing Python’s built-in list.

Example 1: Real-Life Scenario — **To-Do List that Disallows Empty Tasks**

from collections import UserList

class TodoList(UserList):

def append(self, task):

if not task.strip():

raise ValueError("Task cannot be empty!")

super().append(task)

# Usage

todo = TodoList()

todo.append("Buy groceries") # ✅

todo.append("Call mom") # ✅

try:

todo.append(" ") # ❌ Empty task

except ValueError as e:

print(e)

print(todo)

Example 2: NumberList that Only Accepts Integers

from collections import UserList

class NumberList(UserList):

def append(self, item):

if not isinstance(item, int):

raise TypeError("Only integers allowed!")

super().append(item)

# Usage

nums = NumberList()

nums.append(10) # ✅

nums.append(20) # ✅

try:

nums.append("abc") # ❌ Not an integer

except TypeError as e:

print(e)

print(nums)

* **UserString**

UserString is a **wrapper around the built-in string** that makes it easier to create custom string classes.  
It works like a normal string, but you can **add rules or modify behavior** without dealing with the complexities of subclassing str directly.

Example 1: Real-Life Scenario — **String Without Numbers**

from collections import UserString

class NoNumbersString(UserString):

def \_\_init\_\_(self, seq):

# Remove any digits from the string

cleaned = ''.join(ch for ch in seq if not ch.isdigit())

super().\_\_init\_\_(cleaned)

# Usage

text = NoNumbersString("H3ll0 W0rld 2025")

print(text)

Example 2: Always Uppercase String

from collections import UserString

class UpperString(UserString):

def \_\_init\_\_(self, seq):

super().\_\_init\_\_(seq.upper())

# Usage

name = UpperString("amit kumar")

print(name)