



Introduction to Modules

- Modules are like building blocks that help us organize our code and make it more reusable



The Role of Modules

- Modules act as containers for related code, promoting code organization.
- Modules facilitate the reuse of code across different projects, enhancing code reusability.
- Collaboration becomes easier as team members can work on different modules independently.
- Maintenance and debugging are simplified due to the isolated nature of modules.



What Are Modules?

- Modules as self-contained units of code.
- Modules can contain functions, classes, and variables.
- Modules provide an organized way to structure code.



Benefits of Using Modules

- Code modularity for better organization.
- Enhanced code reusability across projects.



Importing Modules

- Importing modules is akin to acquiring specialized tools for specific tasks.
- In Python, this process is incredibly straightforward, can use the **import** keyword.

Examples of Importing Modules

- **Example 1: Importing the math Module**

```
import math  
result = math.sqrt(25)  
print(result) # Output: 5.0
```

Examples of Importing Modules...

- **Example 2: Importing the random Module**

```
import random
number = random.randint(1, 10)
print(number) # Output: A random number between 1 and 10
```

Examples of Importing Modules...

- **Example 3: Importing the datetime Module**

```
import datetime  
today = datetime.date.today()  
print(today) # Output: Current date
```




Availability of External Modules

Introduction to External Modules

Beyond Python's built-in modules, a vast universe of external modules is available. These modules are created by developers worldwide to extend Python's capabilities to diverse domains.



Using Package Managers

- Python offers a seamless mechanism using package managers.
- The most popular package manager is pip – Python's package installer. With pip, you can effortlessly download and install external modules with a single command.



Examples of Popular External Modules

- **Example 1: The requests Module**

The requests module empowers your Python programs to make HTTP requests – an essential skill when working with web APIs or fetching data from the internet. To install it, simply execute `pip install requests`



```
import requests
```

```
response = requests.get('https://www.example.com')
```

```
print(response.status_code) # Output: HTTP status code
```



Examples of Popular External Modules...

- **Example 2: The numpy Module**

For heavy-duty numerical computations, the numpy module is a game-changer. It provides a powerful array data structure and a multitude of mathematical functions. You can bring numpy into your projects with `pip install numpy`.



```
import numpy as np
```

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

```
mean = np.mean(array)
```

```
print(mean) # Output: Mean value of the array
```



Power of Extensibility



- These examples simply scratch the surface of the external modules.
- By studying importing and utilizing external modules, you're equipped to tackle an array of challenges in various domains.



Scenarios for Using Modules

- Modules are like tools in your programming toolbox.
- Let's explore into real-world scenarios where modules prove their worth, simplifying complex tasks and accelerating your coding journey



Scenario 1 – Mathematical Calculations

- When you need to perform advanced mathematical calculations, Python's built-in math module comes to the rescue.
- It offers a wide range of mathematical functions, from trigonometric operations to exponentials.



```
import numpy as np
```

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

```
mean = np.mean(array)
```

```
print(mean) # Output: Mean value of the array
```



Scenario 2 – Handling Dates and Times

- The datetime module is invaluable when working with date and time data.
- It provides functionalities to manipulate dates, calculate time differences, and format output.



```
import datetime
```

```
today = datetime.datetime.now()
```

```
formatted_date = today.strftime('%Y-%m-%d')
```

```
print(formatted_date)  # Output: Current date in YYYY-MM-DD format
```



Scenario 3: Data Analysis

- When data analysis is on the agenda, the pandas module takes the spotlight.
- It simplifies tasks such as data loading, manipulation, and analysis.



```
import pandas as pd
```

```
data = {'Name': ['Alice', 'Bob', 'Carol'],  
        'Age': [25, 30, 28]}
```

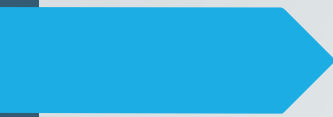
```
df = pd.DataFrame(data)
```

```
print(df) # Output: DataFrame with Name and Age columns
```



Scenario 4: Web Requests

- The requests module facilitates making HTTP requests, making interactions with web services a breeze.



```
import requests
```

```
response = requests.get('https://www.example.com')
```

```
print(response.status_code) # Output: HTTP status code
```




Scenario 5: Graphical Interfaces

- For creating graphical interfaces, the tkinter module offers a range of tools to design user-friendly applications.



```
import tkinter as tk
```

```
root = tk.Tk()
```

```
label = tk.Label(root, text="Hello, World!")
```

```
label.pack()
```

```
root.mainloop()
```



Scenario 6: File Handling

- The os module enables interaction with the operating system, allowing tasks like file handling, directory navigation, and environment queries.



```
import os
```

```
current_directory = os.getcwd()
```

```
print(current_directory) # Output: Current working directory
```



Organizing large programs into functions





Structuring with Functions

- Functions act like building blocks of a program.
- Each function has a specific purpose and can be reused.
- Using functions, we can avoid code repetition and improve code readability.



Task-Oriented Functions



- Functions should have a single task or responsibility.
- This improves code clarity and makes functions easier to test.
- By creating specific functions, we simplify the overall logic of the program.



Reusability and Abstraction



- Functions allow us to reuse code across projects.
- Abstracting tasks into functions conceals complex details.
- This fosters code reuse and speeds up development.



Encapsulation



- Encapsulation means hiding implementation details.
- Functions encapsulate specific tasks, providing a clean interface.
- Encapsulation improves code organization and reduces potential errors



Control Flow and Functions

- Functions enhance control flow in a program.
- We can call functions when needed, maintaining a clear flow.
- This makes our code more structured and comprehensible.



Handling Exceptions

- **The Need for Exception Handling**

Exception handling is crucial to prevent unexpected errors from crashing our programs.

It ensures that our code can gracefully handle issues that may arise during execution.



Exceptions in Modules

- Even modules can raise exceptions if something goes wrong during their operation
- For example, when importing a module, if the module is not found, a 'ModuleNotFoundError' exception occurs.



Handling Module Exceptions

- To handle exceptions raised by modules, we can use 'try...except' blocks.
- This helps our program continue running even if a module-related exception occurs.



Exceptions in Functions

- Functions can also encounter exceptions due to incorrect inputs or unexpected behavior
- For instance, dividing by zero or accessing an index out of bounds in a list can lead to exceptions.



Handling Function Exceptions

- We can use 'try...except' blocks within functions to manage exceptions.
- This way, we can ensure that if an exception occurs, our program doesn't crash but instead executes the specified error-handling code



Exception Hierarchy

- Python's exceptions are organized in a hierarchy.
- At the top is the 'BaseException' class, from which other exception classes are derived.
- Understanding this hierarchy helps us target specific exceptions for handling



Custom Exception Handling

- In addition to built-in exceptions, we can create custom exception classes.
- These classes can represent specific error scenarios in our code.
- This allows us to handle unique errors in a more meaningful way.

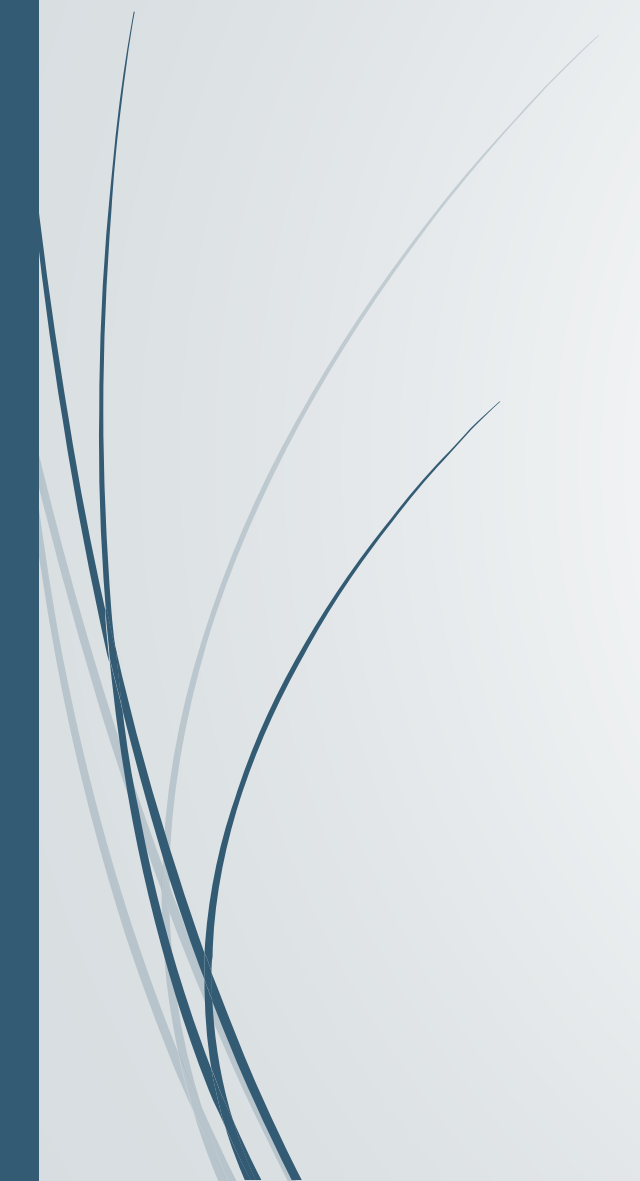
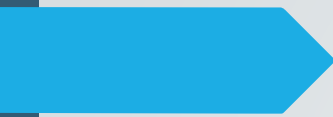
Example of how you can create and use custom exception classes in Python

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount

self.message = f"Insufficient funds. Balance: {balance}, Withdrawal: {amount}"

    def withdraw(balance, amount):
        if amount > balance:
            raise InsufficientFundsError(balance, amount)
        else:
            balance -= amount


print(f"Withdrawal of {amount} successful. New balance: {balance}")
```



```
try:
    account_balance = 500
    withdrawal_amount = 700
    withdraw(account_balance, withdrawal_amount)
except InsufficientFundsError as e:
    print(e.message)
```

In previous example

1. We define a custom exception class `InsufficientFundsError` that inherits from the base `Exception` class.
2. The `__init__` method of the custom exception class takes the current balance and the attempted withdrawal amount as parameters. It constructs an error message with these values.
3. The `withdraw` function simulates a withdrawal operation. If the withdrawal amount is greater than the balance, it raises the `InsufficientFundsError`.



4. In the try block, we create an account balance of 500 and attempt to withdraw 700.

5. When the withdrawal amount exceeds the balance, the custom exception `InsufficientFundsError` is raised.

6. In the except block, we catch the custom exception and print the error message.