Python

Full stack Skills Bootcamp

Introducing Python Error Handling and Exceptions

What is Error Handling?

- Error handling is a critical aspect of programming that allows developers to manage unexpected events gracefully. Exceptions are special conditions that arise during the execution of a program, often indicating an error or an unusual circumstance.

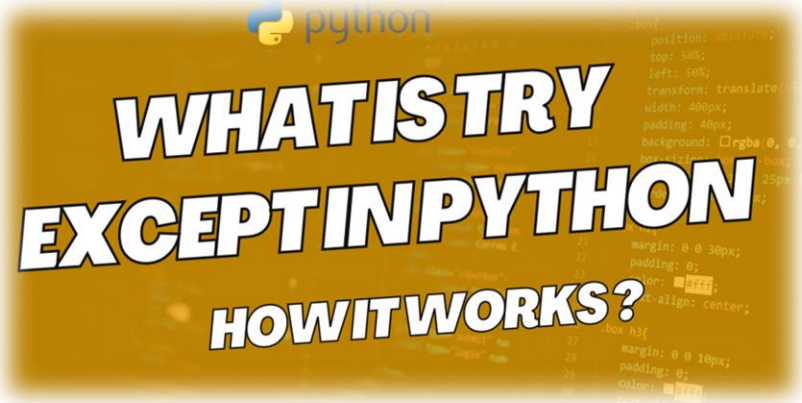
- Importance:

Proper error handling ensures that programs do not crash unexpectedly and can provide informative feedback to users and developers. Understanding how to manage exceptions allows for robust and resilient code.

Basic Try-Except Example

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: You cannot divide by zero!")
```

```
Error: You cannot divide by zero!
```



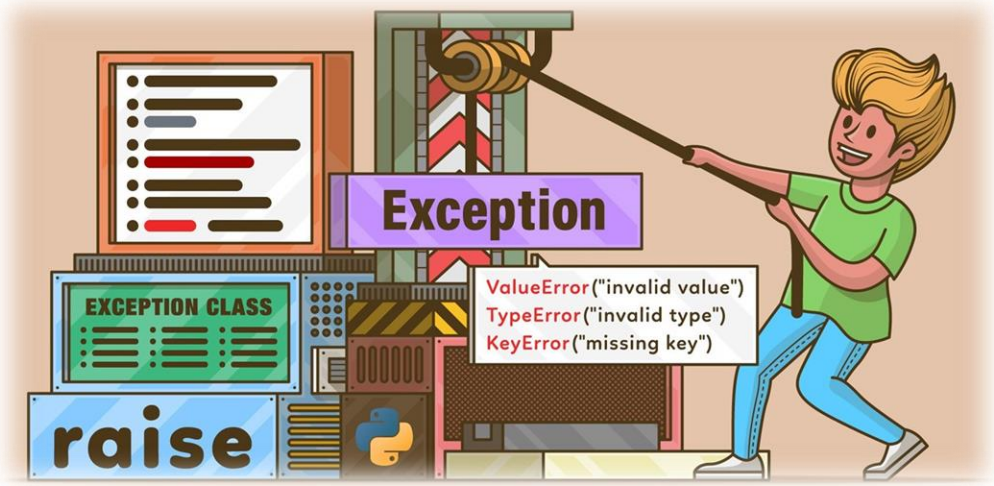
- In this example, attempting to divide by zero raises a ZeroDivisionError. Instead of crashing the program, the error is caught in the except block, allowing you to respond with a user-friendly message.
- This structure helps isolate problematic code and makes it clear where exceptions may occur.

Handling Multiple Exceptions

```python
try:
    number = int("Step8up")
except (ValueError, TypeError) as e:
    print(f"Error: {e}")
```

```
Error: invalid literal for int() with base 10: 'Step8up'
```

- Python allows you to handle multiple exceptions in a single except clause by enclosing them in parentheses.
- Here, converting a string that cannot be interpreted as an integer raises a ValueError. The except block catches both ValueError and TypeError, allowing for concise error handling.
- This feature is useful when you anticipate multiple types of errors that could occur from a single operation.



Using Finally to Execute Code Regardless of an Exception

```
try:
    file = open("test.txt", "r")
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("This will always execute, whether an exception occurs or not.")
```

- The finally block will execute after the try block, regardless of whether an exception was raised or not. This is especially useful for cleanup operations, such as closing files or releasing resources.
- In this example, if the specified file does not exist, a FileNotFoundError is caught, and a message is printed. Regardless of the error, the finally block ensures that a closing message is always displayed.

Using Else Block

```python
try:
    number = 5
    result = number / 2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print(f"The result is {result}.")
```

```
The result is 2.5
```

- This slide illustrates the use of the else block for handling successful operations. If the division succeeds, the result is printed.
- This structure allows for clearer logic in your code, distinguishing between error cases and regular flow without nesting.

Raising Custom Exceptions

```python
age = -1
if age < 0:
    raise ValueError("Age cannot be negative!")
```

```
Traceback (most recent call last):
  ...
ValueError: Age cannot be negative!
```

- You can raise exceptions intentionally using the raise statement. This is particularly useful for enforcing rules and constraints within your code.
- This example checks the validity of the age variable. If it's negative, a ValueError is raised, halting execution and providing a clear error message.

Exception Example with Detailed Logging

```python
try:
    my_list = [1, 2, 3]
    print(my_list[5])  # This will raise an IndexError
except IndexError:
    print("Error: Index out of range.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

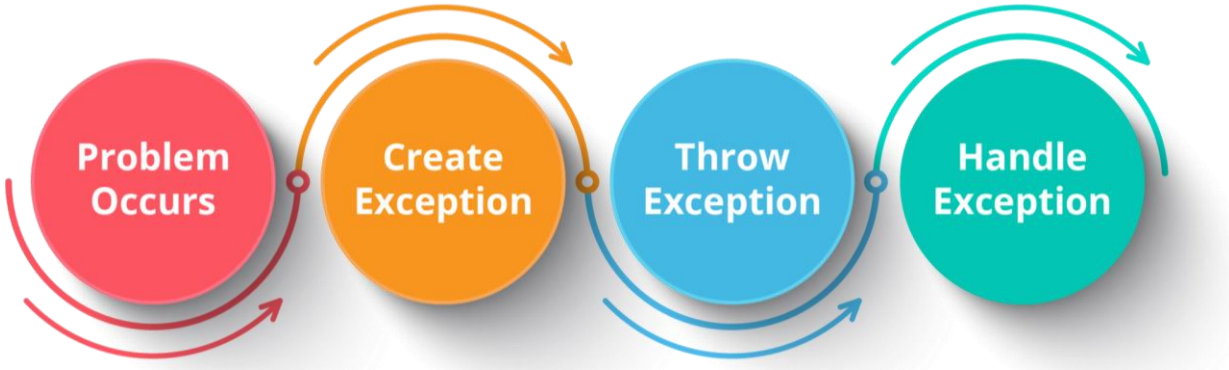
```
Error: Index out of range.
```

- Detailed error logging helps developers understand what went wrong and where, improving debugging efforts and application maintenance.
- This example highlights how catching specific exceptions, like IndexError, can lead to clearer, more informative messages. The logging of the error gives insights into what caused the issue, aiding in troubleshooting.
- By catching different types of exceptions separately, you gain better control over how to respond to each situation.

Concluding Error Handling

Key Points

- Always use specific exceptions to catch and handle errors effectively. This prevents unintended behaviours and simplifies debugging.
- Avoid using broad except clauses, as they can mask real problems in the code.
- Consider implementing logging to capture exception details, which aids in understanding issues when they arise.
- Always clean up resources (like files and network connections) in the finally block to maintain system integrity.



Introducing Python Classes

What is Class?

- A class is a fundamental concept in Object-Oriented Programming (OOP). It acts as a template or blueprint for creating objects.
- Attributes represent the data that describes the object.
- Methods are functions defined inside the class that define the object's behaviour.

__init__(self, name, age):

- The constructor method used to initialize the attributes name and age when a new object is created.
- The self-parameter refers to the current instance of the class. It is how the object keeps track of its attributes and methods.

Attributes like name and age define the state of the object.

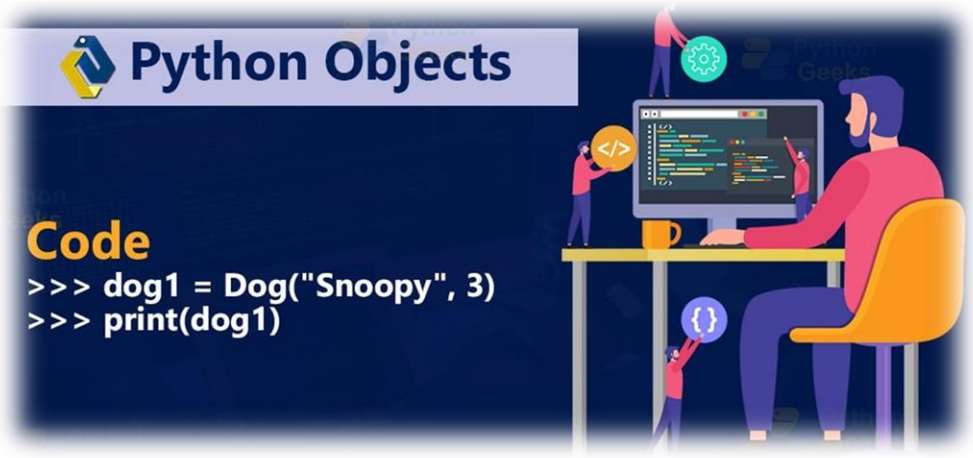
Methods like bark() and get_age() define what actions the object can perform.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name  # Attribute
        self.age = age    # Attribute

    def bark(self):
        return f"{self.name} says woof!"

    def get_age(self):
        return f"{self.name} is {self.age} years old."
```

Objects can have different values for their attributes.

Creating Objects

```python
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)
```



- An object is a specific instance of a class. It represents a real-world entity created using the class as a template. Every object has its own set of attributes, which are initialized through the class constructor.

- Each object can have unique values for its attributes, even though both objects are instances of the same class.

- When creating an object, the __init__ method is automatically called to initialize the object's attributes.

Accessing Attributes and Methods

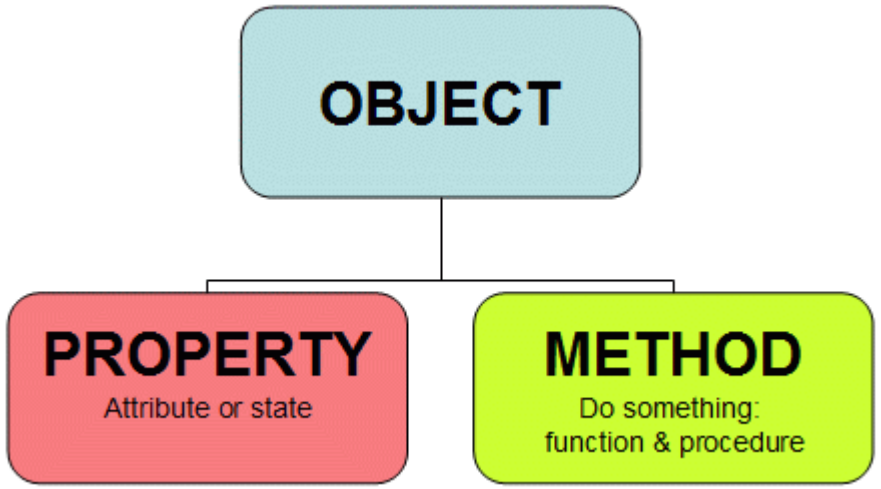
```python
print(dog1.get_age())  # Output: Buddy is 3 years old.
print(dog1.bark())     # Output: Buddy says woof!
```

- You can call methods of an object using the dot notation.

Calling Methods:

- dog1.get_age() accesses the get_age() method of the dog1 object, returning a string with the dog's name and age.
- dog1.bark() invokes the bark() method of dog1, returning the string "Buddy says woof!".

Dot Notation: In Python, you use the dot notation (.) to access methods or attributes of an object.

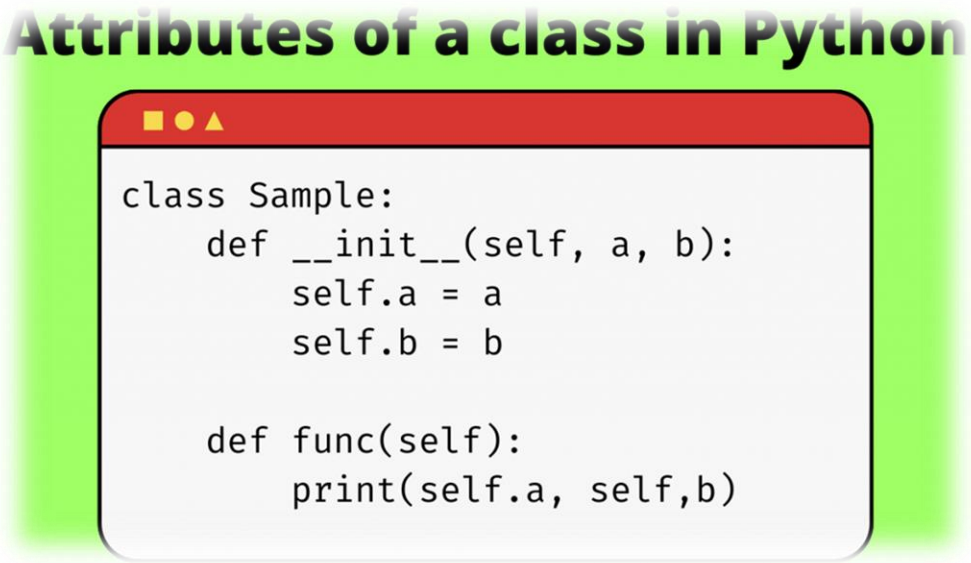


Modifying Attributes

```python
dog1.age = 4
print(dog1.get_age())  # Output: Buddy is 4 years old.
```

- After creating an object, you can directly modify the object's attributes by assigning new values.

- The attribute age of dog1 was originally set to 3 when the object was created.
- Now, we've modified the age attribute of dog1 to 4. When we call dog1.get_age() again, it reflects the updated age.
- Modifying attributes allows us to change the internal state of an object dynamically after creation.



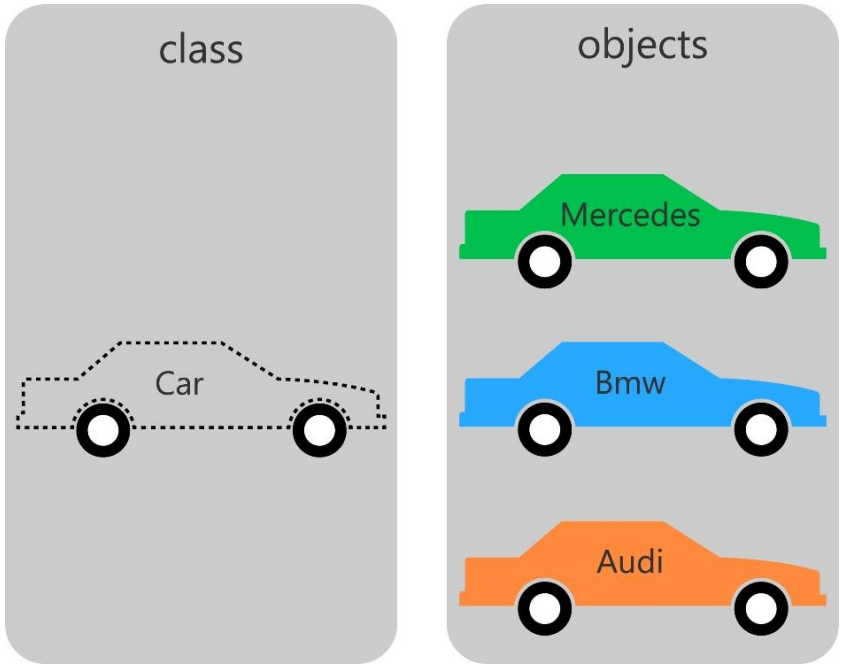
Real-World Analogy

Class as a Blueprint:

- Think of a class as a blueprint for creating objects. For example, a blueprint for a car contains details like the car's design, components, and features, but it isn't an actual car.

Object as a Physical Entity:

- When the blueprint is used to create an actual car, that car is an object. Multiple cars (objects) can be created from the same blueprint (class), but each car may have its own unique features (attributes), like color, engine type, etc.



Object-Oriented Programming (OOP)

Reusability: You can create multiple objects from the same class, reducing code duplication.

- Example: You can define the Dog class once and create multiple dog objects (dog1, dog2).

Modularity: Classes help you organize code into logical units.

- Example: A Car class can have separate classes for Engine or Wheel, making the code modular.

Abstraction: Focus on using the object's methods rather than its internal implementation.

- Example: You just call dog1.bark(), without worrying about how the bark() method is implemented.

Concluding OOPS

Key Points

- Classes provide a structured way to define entities, allowing you to define their attributes and behavior.
- Objects are specific instances of classes, representing unique entities with their own data.
- Object-Oriented Programming (OOP) principles like encapsulation, abstraction, and reusability help organize and optimize your code for scalability.