Exception Handling

Exception handling in Python allows you to handle errors and exceptions that may occur during the execution of your program. It helps you gracefully handle these errors and provide appropriate actions or messages to the user. The basic syntax for exception handling in Python is the `try-except` block.

Here's an example that demonstrates exception handling in Python:

```python
try:
   # Code that may raise an exception
   num1 = int(input("Enter a number: "))
   num2 = int(input("Enter another number: "))
   result = num1 / num2
   print("The result is:", result)

except ValueError:
   print("Invalid input. Please enter a valid number.")

except ZeroDivisionError:
   print("Cannot divide by zero.")
```

```
except Exception as e:
    print("An error occurred:", str(e))

else:
    print("No exceptions occurred.")

finally:
    print("Exception handling is complete.")
```

In the above example, we have a `try` block where we put the code that may raise an exception. If an exception occurs, it is handled in the respective `except` block based on the type of exception. In this case, we handle `ValueError` when the user enters an invalid number, and `ZeroDivisionError` when the user attempts to divide by zero.

The `except` block with the `Exception` keyword acts as a catch-all block, which will handle any other exceptions not caught by the previous `except` blocks. The `as` keyword allows you to assign the exception to a variable (`e` in this case), so you can access information about the exception if needed.

The `else` block is executed if no exceptions occur in the `try` block, and the `finally` block is always executed, regardless of whether an exception occurred or not.

Here's another example that demonstrates how to raise and handle a custom exception:

```python
class CustomException(Exception):
    pass


try:
    age = int(input("Enter your age: "))


    if age < 0:
        raise CustomException("Age cannot be negative.")


    print("Your age is:", age)


except CustomException as ce:
    print("Custom exception caught:", str(ce))


except Exception as e:
    print("An error occurred:", str(e))
```

In this example, we define a custom exception `CustomException` by creating a new class that inherits from the `Exception` class. We use the `raise` keyword to raise an instance of this custom exception if the user enters a negative age. The raised exception is then caught in the corresponding `except` block.

By using exception handling, you can ensure that your program handles errors gracefully and provides informative messages to the user, enhancing the overall user experience and making your code more robust.

Exception Handling Statements

In Python, exception handling allows you to handle and manage errors that may occur during the execution of your code. You can use various exception handling statements to catch and respond to different types of exceptions. Here are the main exception handling statements in Python:

1. **try-except**: The `try` block is used to enclose the code that might raise an exception. If an exception occurs within the `try` block, it is caught and handled by the corresponding `except` block. The `except` block specifies the type of exception to catch and defines the actions to be taken.

```python
try:
    # Code that might raise an exception
except ExceptionType:
    # Exception handling code
```

You can also catch multiple exceptions by specifying them as a tuple:

```python
try:
    # Code that might raise an exception
except (ExceptionType1, ExceptionType2):
    # Exception handling code
```

2. **else**: The `else` block is used in conjunction with the `try-except` statement. It is executed if no exceptions are raised in the `try` block. This block is optional.

```python
try:
    # Code that might raise an exception
except ExceptionType:
    # Exception handling code
else:
```

    # Code executed if no exception occurs

3. **finally**: The `finally` block is used to define cleanup actions that must be executed, whether an exception occurs or not. It is executed regardless of whether an exception was caught or propagated.

```python
try:
    # Code that might raise an exception
except ExceptionType:
    # Exception handling code
finally:
    # Code that always runs, regardless of exceptions
```

4. **raise**: The `raise` statement allows you to manually raise an exception. You can use it to create your custom exceptions or re-raise exceptions that you caught.

```python
raise ExceptionType("Error message")
```

5. **assert**: The `assert` statement is used to test if a given condition is true. If the condition is false, it raises an `AssertionError` exception. It is commonly used for debugging and sanity checks.

```python
assert condition, "Error message"
```

These are the fundamental exception handling statements in Python. By using them effectively, you can handle exceptions gracefully and control the flow of your program when errors occur.

Turtle Programming in Python

Turtle graphics is a popular module in Python that allows you to create drawings and animations using a virtual turtle. The turtle can be moved around the screen and instructed to draw lines of various lengths and angles. Here's an explanation of turtle graphics programming with examples:

1. **Importing the turtle module**: First, you need to import the `turtle` module to use its functions and classes.

```python
```

import turtle

2. **Creating a turtle object**: Next, you create a turtle object that represents the virtual turtle on the screen.

```python

my_turtle = turtle.Turtle()

```

3. **Moving the turtle**: You can move the turtle using various functions. Here are a few examples:

  - `forward(distance)`: Moves the turtle forward by the specified distance.

  - `backward(distance)`: Moves the turtle backward by the specified distance.

  - `right(angle)`: Turns the turtle right by the specified angle.

  - `left(angle)`: Turns the turtle left by the specified angle.

```python

my_turtle.forward(100)

my_turtle.right(90)

my_turtle.forward(50)
```

4. **Drawing lines**: To draw lines, you can use the `forward()` function. You can specify the length of the line, and the turtle will move forward by that distance, leaving a trail.

```python
my_turtle.forward(200)
```

5. **Changing the turtle's appearance**: You can modify the turtle's appearance using various functions. For example:

   - `color("color_name")`: Sets the turtle's color.
   - `pensize(width)`: Sets the width of the turtle's pen.
   - `shape("shape_name")`: Changes the turtle's shape.

```python
my_turtle.color("red")
my_turtle.pensize(3)
my_turtle.shape("turtle")
```

6. **Controlling the pen**: The turtle has a pen that can be raised or lowered. When the pen is down, the turtle will leave a trail as it moves.

   - `penup()`: Lifts the pen, so the turtle doesn't draw.
   - `pendown()`: Lowers the pen, so the turtle draws.

```python
my_turtle.penup()
my_turtle.goto(100, 100)
my_turtle.pendown()
my_turtle.goto(200, 200)
```

7. **Complete example**: Here's an example that draws a square using turtle graphics:

```python
import turtle

my_turtle = turtle.Turtle()

for _ in range(4):
    my_turtle.forward(100)

    my_turtle.right(90)


turtle.done()
```

In this example, the turtle moves forward by 100 units and turns right by 90 degrees four times, creating a square shape.

These are the basic concepts and functions of turtle graphics programming in Python. By combining movement, drawing, and other turtle commands, you can create more complex drawings and animations.

File handling in python

File handling in Python allows you to work with files on your computer's file system. It enables you to read data from files, write data to files, and perform various file operations. Here's an explanation of file handling in Python with examples:

1. **Opening a file**: To work with a file, you need to open it using the `open()` function. It takes two parameters: the file name and the mode (read, write, append, etc.).

```python
file = open("myfile.txt", "r")
```

2. **Reading from a file**: Once a file is opened, you can read its contents using various methods. Here are a few examples:

 - `read()`: Reads the entire content of the file as a single string.
 - `readline()`: Reads a single line from the file.
 - `readlines()`: Reads all lines from the file and returns them as a list of strings.

```python
file = open("myfile.txt", "r")
content = file.read()
print(content)

file = open("myfile.txt", "r")
line = file.readline()
print(line)

file = open("myfile.txt", "r")
lines = file.readlines()
```

```
print(lines)
```

3. **Writing to a file**: To write data to a file, you need to open it in write mode (`"w"`). You can use the `write()` method to write a string to the file.

```python
file = open("myfile.txt", "w")
file.write("Hello, World!")
file.close()
```

4. **Appending to a file**: If you want to add content to an existing file without overwriting it, you can open the file in append mode (`"a"`). The `write()` method will then add the new data to the end of the file.

```python
file = open("myfile.txt", "a")
file.write("Appending a new line!")
file.close()
```

5. **Closing a file**: It's important to close a file after you finish working with it. You can do this using the `close()` method. It ensures that all changes are saved and frees up system resources.

```python
file = open("myfile.txt", "r")
# Perform file operations
file.close()
```

6. **Using a context manager (with statement)**: An alternative and recommended approach is to use a context manager (`with` statement). It automatically takes care of closing the file, even if an exception occurs.

```python
with open("myfile.txt", "r") as file:
    content = file.read()
    print(content)
```

7. **Complete example**: Here's an example that demonstrates reading and writing to a file:

```python
# Writing to a file
with open("myfile.txt", "w") as file:
    file.write("Hello, World!")

# Reading from the file
with open("myfile.txt", "r") as file:
    content = file.read()
    print(content)
```

In this example, the file "myfile.txt" is created and the string "Hello, World!" is written to it. Then, the content of the file is read and printed.

These are the basic operations involved in file handling in Python. By combining these operations, you can manipulate files, read data, write data, and perform other file-related tasks in your Python programs.