**Download Your Daily Notes**

**My own views on the usefulness of understanding errors and exceptions**

 Today's note focuses on syntax errors and exceptions in Python programming. Syntax errors, also known as parsing errors, are common when learning Python. They occur when there is an issue with the code structure. The error message indicates the line and file where the error was detected.  
  
Exceptions, on the other hand, occur during program execution and can be handled in Python programs. They are not fatal and can be managed to prevent program termination. Different types of exceptions exist, such as ZeroDivisionError, NameError, and TypeError, each indicating a specific error type.  
  
The note also discusses how to handle exceptions using the try-except statement. The try clause is used to execute the code, and if an exception occurs, it is caught by the except clause. Multiple except clauses can be used to handle different exceptions. The note provides examples of exception handling and explains how to use exception classes and the optional else clause.  
  
Additionally, the note mentions that exceptions can have associated values or arguments. These arguments can be accessed and used in exception handling. The note concludes by highlighting that exception handlers can handle exceptions occurring not only in the try clause but also inside functions called within it.

**Daily Notes - Errors and Exceptions**

 Today's note focuses on syntax errors and exceptions in Python programming. Syntax errors, also known as parsing errors, are common when learning Python. They occur when there is an issue with the code structure. The error message indicates the line and file where the error was detected.  
  
Exceptions, on the other hand, occur during program execution and can be handled in Python programs. They are not fatal and can be managed to prevent program termination. Different types of exceptions exist, such as ZeroDivisionError, NameError, and TypeError, each indicating a specific error type.  
  
The note also discusses how to handle exceptions using the try-except statement. The try clause is used to execute the code, and if an exception occurs, it is caught by the except clause. Multiple except clauses can be used to handle different exceptions. The note provides examples of exception handling and explains how to use exception classes and the optional else clause.  
  
Additionally, the note mentions that exceptions can have associated values or arguments. These arguments can be accessed and used in exception handling. The note concludes by highlighting that exception handlers can handle exceptions occurring not only in the try clause but also inside functions called within it.

**Daily Notes - Raising an Exception**

 In today's note, we cover several aspects of exception handling in Python. We start by exploring how to raise an exception using the raise statement. It allows us to throw a custom exception when a specific condition occurs. An example is provided where an exception is raised if a variable exceeds a certain value.  
  
Next, we learn about assertions in Python. Assertions are used to check if a condition is met, and if not, an AssertionError exception is raised. The code snippet demonstrates the use of assertions to validate temperature values.  
  
The note then introduces the try-except block, which is used to catch and handle exceptions in Python. The code within the try block is executed normally, and if an exception occurs, the code in the corresponding except block is executed. An example is given where file handling and error handling are combined using the try-except block.  
  
The finally block is discussed, which is an optional part of the try-except statement. The code in the finally block is always executed, regardless of whether an exception occurred or not. Examples are provided to demonstrate its usage.  
  
Moving on, the note mentions user-defined exceptions. Python allows creating custom exceptions by deriving classes from built-in exceptions. An example related to the RuntimeError class is given, showcasing how to define and handle custom exceptions.  
  
Lastly, the note explains clean-up actions using the try-finally construct. The finally block is executed as the last task before the try statement completes, even if an exception occurs. Different scenarios and behaviors related to exceptions and the finally block are discussed with examples.  
  
By understanding these concepts, you can effectively handle exceptions in Python and control the flow of your programs.

**Daily Notes - Activity 1 - Explaining Exceptions**

 Title: Understanding Exceptions in Python  
  
[Slide 1: Introduction]  
Welcome, everyone! Today, we're going to dive into the world of exceptions in Python. Exceptions are an essential part of writing reliable and robust code. They help us handle errors and exceptional conditions that may occur during program execution.  
  
[Slide 2: What are Exceptions?]  
Exceptions are events that disrupt the normal flow of a program. They are raised when an error or exceptional condition is encountered. Instead of crashing the program, exceptions provide a way to handle and recover from errors, making our code more resilient.  
  
[Slide 3: How Exceptions Work]  
Let's take a look at the basic workflow of exceptions:  
An error occurs during program execution.  
An exception object is created, containing information about the error.  
The program stops its normal execution and looks for exception handlers.  
If a matching exception handler is found, the error is handled, and program execution resumes.  
If no handler is found, the program terminates and displays an error message.  
  
[Slide 4: Common Exception Types]  
Python provides various built-in exception types. Here are some common ones you may encounter:  
SyntaxError: Occurs when the code violates the language syntax rules.  
NameError: Raised when a local or global name is not found.  
TypeError: Occurs when an operation is performed on incompatible data types.  
ValueError: Raised when a function receives an argument of the correct type but with an invalid value.  
FileNotFoundError: Raised when a file or directory is not found.  
Understanding these exception types helps us identify and handle specific errors in our programs effectively.  
  
[Slide 5: Exception Handling]  
Exception handling allows us to gracefully handle exceptions, preventing our programs from crashing abruptly. It provides an opportunity to recover from errors or take alternative actions.  
  
[Slide 6: Try-Except Block]  
The primary construct for handling exceptions in Python is the try-except block. It has the following structure:  
  
try:  
# Code that may raise an exception  
except ExceptionType:  
# Exception handling code  
finally:  
# Optional cleanup code  
The code that could potentially raise an exception is enclosed within the try block. The except block specifies the exception type(s) to catch and the corresponding error handling code. You can use multiple except blocks to handle different exceptions. Additionally, you can include a finally block for optional cleanup actions.  
  
[Slide 7: Exception Handling Example]  
Let's see an example of how exception handling works:  
  
python  
Copy code  
try:  
# Code that may raise an exception  
result = 10 / 0 # Division by zero error  
except ZeroDivisionError:  
# Handling the specific exception  
print("Error: Division by zero occurred!")  
finally:  
# Optional cleanup code  
print("Cleanup actions here")  
  
print("Continuing with the rest of the program...")  
In this example, when the division by zero error occurs, it is caught by the except block, which handles the exception and executes the cleanup code in the finally block. The program continues its execution afterward.  
  
[Slide 8: Exception Propagation]  
Exceptions can propagate through the call stack. If an exception is not handled within a function, it is passed to the calling function. If the exception reaches the top-level of the program without being handled, it results in program termination.  
  
[Slide 9: Raising Exceptions]  
In addition to handling exceptions, we can also raise exceptions explicitly using the raise statement. This allows us to signal exceptional conditions in our code. Here's an example:  
  
python  
Copy code  
if condition:  
raise CustomException("Error message")  
We can raise custom exceptions by creating our own exception classes and providing appropriate error messages.  
  
[Slide 10: Benefits of Exception Handling]  
Let's summarize the benefits of using exception handling in our programs:  
  
Improved program reliability and robustness by handling errors gracefully.  
Error recovery and the ability to take alternative actions.  
Enhanced code readability by separating error handling logic from the main code.  
Easier debugging and troubleshooting, as exceptions provide valuable error information.  
[Closing Slide]  
Exception handling is a powerful tool for writing more reliable and maintainable Python code. By understanding how exceptions work and incorporating proper handling techniques, we can create more resilient and user-friendly applications.  
  
Thank you for joining me in this short presentation on exceptions in Python.

**Daily Notes - Activity 2 - Applying new Concepts**

 We will not be implementing this on our school website.