Python:
Exceptions

Defensive Programming

- Defensive programming is a software development approach that focuses on writing code that is resilient to errors, robust in handling exceptional conditions, and less prone to failures.
- The goal of defensive programming is to anticipate and prevent potential issues and ensure that the program can handle unexpected situations gracefully. It involves techniques and practices that prioritize reliability, fault tolerance, and maintainability.

cont...

- One of the policies of defensive programming is **exception handling**
- It involves writing code that anticipates potential issues and handles them proactively. Validate assumptions, check preconditions, and add assertions or guard clauses to ensure code correctness and prevent unexpected behavior.

What are exceptions?

- Since python is an interpreted language, a python program will terminate the moment it spots an error.

- In Python, an error can be a syntax error or an exception.
  - **Syntax error** is an error that occurs when the parser detects an incorrect statement.
  - **An exception** is a type of error that occurs whenever syntactically correct Python code results in an error.

Cont..

- The goal of exceptions is to catch an error we anticipated and produce a correct error output to the end user.
- Their are different exceptions that can be used to help in specifying the error.
- Some include, IndexError, ValueError, Zerodivisionerror, NameError ….
- The goal of these exceptions is to give a comprehensive description of what went wrong in the program.

Examples...

- **What happens when procedure execution hits an unexpected condition?**

- get an exception... to what was expected
- trying to access beyond list limits

  test = [1,7,4]

  test[4] → IndexError

- trying to convert an inappropriate

  type int(test) → TypeError
- referencing a non-existing variable

  a → NameError

- mixing data types without coercion

  'a'/4 → TypeError

Specific Exception in use:

```
def safe_print_list(my_list=[], x=0):
    try:
        a = 0
        for i in range(x):
            print(my_list[i], end="")
            a += 1
        print("")
        return x
    except IndexError:
        print("")
        return a
```

Example error:

Syntax Error: You can only handle this by debugging

```
>>> for i in range(10)
  File "<stdin>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
>>>
```

Chaining Exceptions

- Exceptions can be chained together to specify handlers for different exceptions.

```
>>> def evenodd(num):
...     try:
...             if num % 2 == 0:
...                     return 'even'
...             else:
...                     return 'odd'
...     except TypeError:
...             return 'Number only'
...     except:
...             return 'General Error'
```

try...except...else:

- The try...except...else block is used when you want to specify code that should only execute if no exceptions occur. It provides an alternative path of execution when the try block succeeds without raising any exceptions. If an exception is raised within the try block, the corresponding except block is executed, and the code in the else block is skipped. The else block is optional and can be omitted if not needed.

try...except...finally

- The try...except...finally block is used when you want to specify code that should always execute, regardless of whether an exception occurs or not. The finally block is executed regardless of whether an exception is raised and caught. It is typically used for cleanup actions such as closing files, releasing resources, or restoring the program state. The finally block is optional, but if present, it will always be executed, even if there is a return statement or an unhandled exception.

Cont

- Beyond chaining exceptions, there are other actions we can do after an exception:
  - Like an else statement that runs after the final exception.

```
def concat(str1, str2):
    try:
            return str1 + str2
    except TypeError:
            return "String only"
    else:
            return "Statment"
```

  - Or a finally block that always runs in the try/except clause, regardless of the exception,

```
def divideZero(num):
    try:
            return num // 0
    except ZeroDivisionError:
            return "Num cant be divided by zero"
    else:
            return "General error statment"
    finally:
            return num
```

Raise exception

- If you want to call an exception error without an actual exception occurring, the raise statement is used.
  - It allows us to raise an exception of any type be it typeerror or zerodivisionerror, as long as it is a subclass of Exception class, and pass a value to use as a error message,
  - We can also raise an exception inside an except clause, leading to more than one exception being outputted.

```
def divideZero(num):
    try:
            return num // 0
    except ZeroDivisionError:
            raise ValueError("num can't be divided")
```

- In this updated code, the except block catches the ZeroDivisionError exception. Instead of directly raising a ValueError, we use the raise ... from ... syntax to raise a new ValueError exception while preserving the original exception as the cause.
- By adding from e at the end of the raise statement, the original ZeroDivisionError exception is chained to the new ValueError exception. This chaining helps provide additional information about the root cause of the exception.

Conclusion

- Exceptions are a powerful tool used to handle errors in our code,
- They can be used in a general sense or for a specific error type,
- They can be chained together to handle multiple types of errors,
- They can have a finally clause which runs regardless of the exception,
- We can use the raise statement to call specific exceptions even when an error hasn't occurred.
- We can use raise inside an exception block to chain exceptions together.

See you at the next session!