PW SKILLS

General use of exception handling

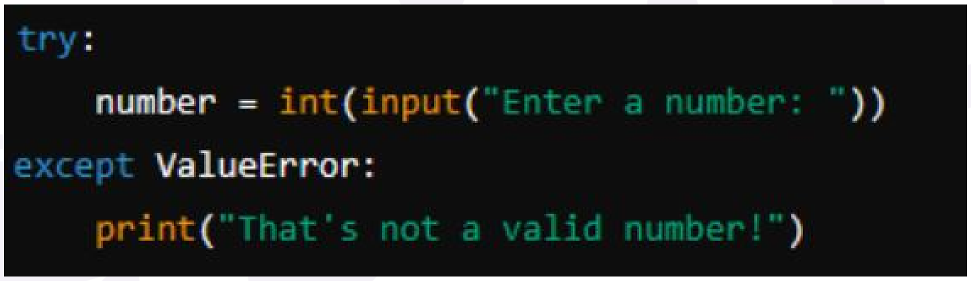
Lesson Plan

Why should you use Exceptions?
Here are the reasons for using exceptions in Python:

- Exception handling in python allows you to separate error-handling code from normal code.
- An exception is a Python object which represents an error.
- As with code comments, exceptions help you to remind yourself of what the program expects.
- It clarifies the code and enhances readability.
- Allows you to stimulate consequences as the error-handling takes place in one place and in one manner.
- An exception is a convenient method for handling error messages.
- In Python, you can raise an exception in the program by using the raise exception method.
- Raising an exception helps you to break the current code execution and returns the exception back to expectation until it is handled.
- Processing exceptions for components that can't handle them directly.

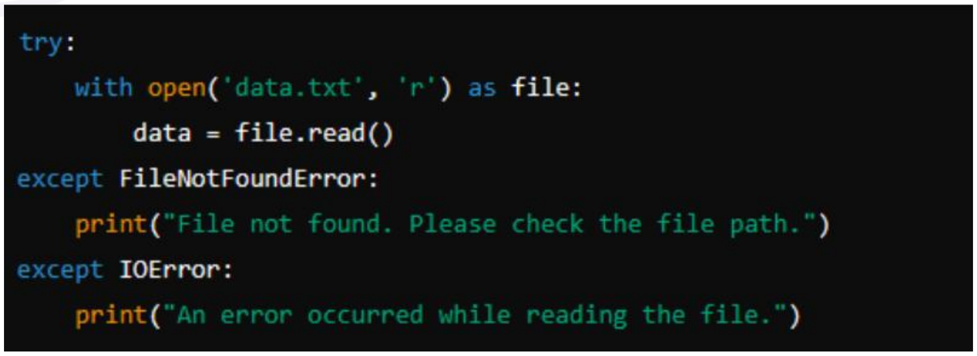
General use cases:

1. Handling Common Errors
When working with user input, file operations, or network requests, errors are common. For example, a user might input a string when your program expects an integer:



```python
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That's not a valid number!")
```

2. File Handling
File operations often involve checking if a file exists, handling read/write errors, or dealing with unexpected formats:



```python
try:
    with open('data.txt', 'r') as file:
        data = file.read()
except FileNotFoundError:
    print("File not found. Please check the file path.")
except IOError:
    print("An error occurred while reading the file.")
```

3. Network Operations
Network operations are prone to errors like timeouts, connection issues, etc. Exception handling ensures your program can respond to these issues appropriately:

```python
import requests

try:
    response = requests.get('https://example.com')
    response.raise_for_status()  # Raises an HTTPError for bad responses
except requests.exceptions.HTTPError as http_err:
    print(f"HTTP error occurred: {http_err}")
except requests.exceptions.ConnectionError as conn_err:
    print(f"Connection error occurred: {conn_err}")
except requests.exceptions.Timeout as timeout_err:
    print(f"Timeout error occurred: {timeout_err}")
except Exception as err:
    print(f"An error occurred: {err}")
```

4. Graceful Shutdowns
In long-running applications like servers or services, it's crucial to handle exceptions to allow for a graceful shutdown, releasing resources, and saving data:

```python
try:
    # Some long-running operation
except KeyboardInterrupt:
    print("Interrupted by user, shutting down.")
finally:
    clean_up_resources()
```

5. Custom Exceptions
Sometimes, you might need to define your own exceptions for specific errors in your application:

```python
class CustomError(Exception):
    pass

def do_something(x):
    if x < 0:
        raise CustomError("Negative value is not allowed")
    return x

try:
    do_something(-1)
except CustomError as e:
    print(e)
```