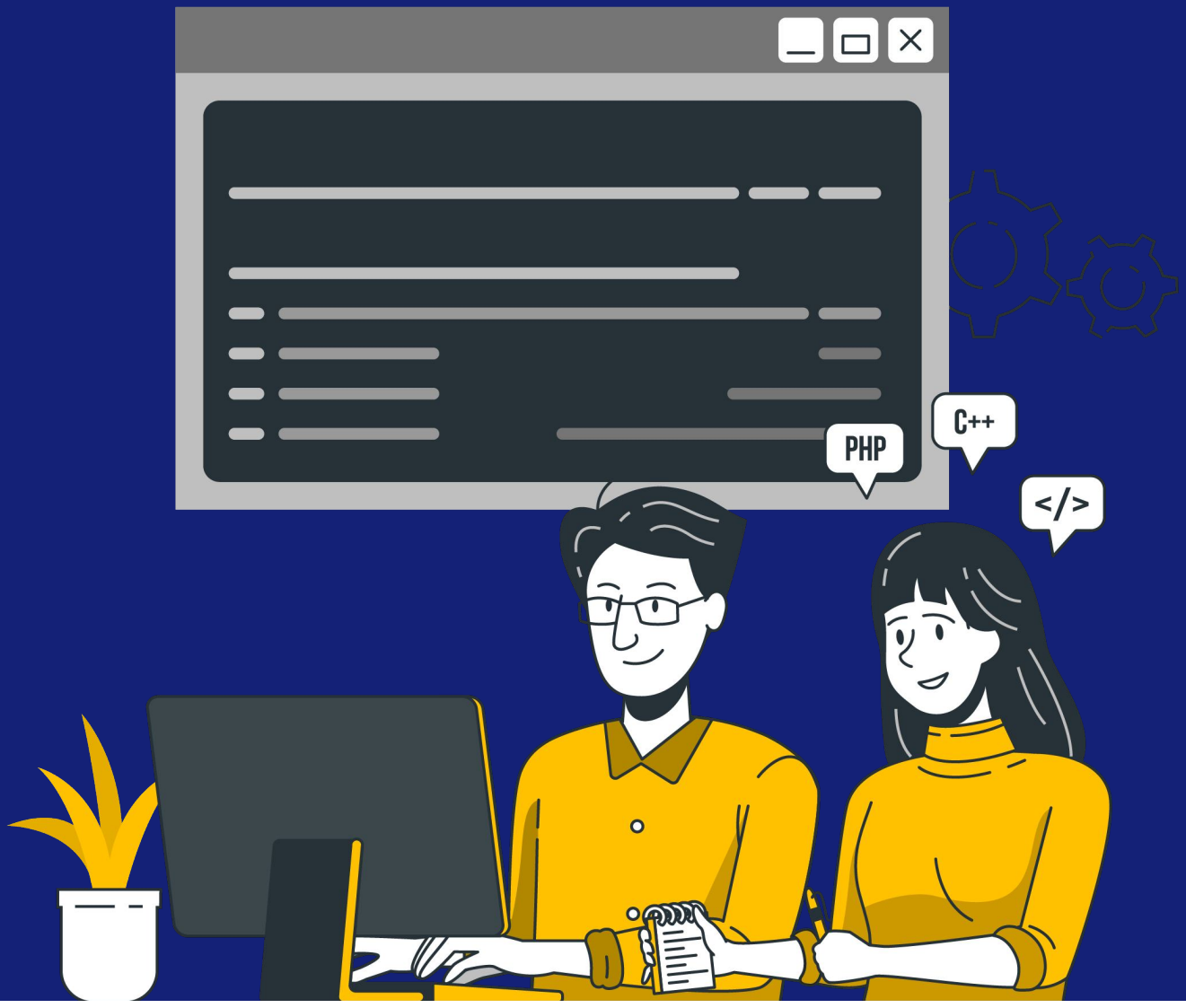
PW SKILLS

Exception Handling

Lesson Plan



In Python, exception handling is a way to manage errors that may occur during the execution of your code. It helps to prevent the program from crashing and allows you to handle errors gracefully.

What is Exception Handling in Python?

Exception handling in Python is a process of resolving errors that occur in a program. This involves catching exceptions, understanding what caused them, and then responding accordingly. Exceptions are errors that occur at runtime when the program is being executed. They are usually caused by invalid user input or code that is invalid in Python. Exception handling allows the program to continue to execute even if an error occurs.

Basic syntax:

```
try:
    # Code that may cause an exception
except SomeException as e:
    # Code to handle the exception
else:
    # Code to run if no exceptions occur
finally:
    # Code that runs no matter what (optional)
```

Key Components

- **try:** The block of code where you write the code that might cause an exception.
- **except:** This block catches the exception and lets you handle it. You can specify the type of exception (like ValueError, TypeError, etc.), or use a general except to catch any exception.
- **else:** This block runs if no exceptions are raised in the try block.
- **finally:** This block always runs, whether an exception was raised or not. It's often used for cleanup actions.

Different Types of Exceptions in Python

1. **NameError**: This Exception is raised when a name is not found in the local or global namespace.
2. **IndexError**: This Exception is raised when an invalid index is used to access a sequence.
3. **KeyError**: This Exception is thrown when the key is not found in the dictionary.
4. **ValueError**: This Exception is thrown when a built-in operation or function receives an argument of the correct type and incorrect value.
5. **IOError**: This Exception is raised when an input/output operation fails, such as when an attempt is made to open a non-existent file.
6. **ImportError**: This Exception is thrown when an import statement cannot find a module definition or a from ... import statement cannot find a name to import.
7. **SyntaxError**: This Exception is raised when the input code does not conform to the Python syntax rules.
8. **TypeError**: This Exception is thrown when an operation or function is applied to an object of inappropriate type
9. **AttributeError**: occurs when an object does not have an attribute being referenced, such as calling a method that does not exist on an object.
10. **ArithmeticError**: A built-in exception in Python is raised when an arithmetic operation fails. This Exception is a base class for other specific arithmetic exceptions, such as ZeroDivisionError and OverflowError.
11. **Floating point error:** It is a type of arithmetic error that can occur in Python and other programming languages that use floating point arithmetic to represent real numbers.
12. **ZeroDivisionError:** This occurs when dividing a number by zero, an invalid mathematics operation.
13. **FileExistsError:** This is Python's built-in Exception thrown when creating a file or directory already in the file system.
14. **PermissionError:** A built-in exception in Python is raised when an operation cannot be completed due to a lack of permission or access rights.

Example

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
else:
    print("The division was successful")
finally:
    print("This will always run")
```

In this example:

- If you try to divide by zero, a **ZeroDivisionError** will be raised, and the **except** block will handle it by printing an error message.
- The **finally** block will always run, regardless of whether an exception was raised or not.

Handling Multiple Exceptions
You can handle multiple exceptions by specifying them in a tuple:

```
try:
    # some code
except (TypeError, ValueError) as e:
    print(f"An error occurred: {e}")
```

Raising Exceptions
You can also raise exceptions manually using the raise keyword:

```
if not isinstance(x, int):
    raise TypeError("Only integers are allowed")
```

This lets you control when and why exceptions are raised in your code.