Error Handling Review

Common Errors in Python

- **SyntaxError**: The error Python raises when your code does not follow its syntax rules. For example, the code `print("Hello there"` will lead to a syntax error with the message, `SyntaxError: '(' was never closed`, because the code is missing a closing parenthesis.
- **NameError**: Python raises a `NameError` when you try to access a variable or function you have not defined. For instance, if you have the line `print(username)` in your code without having a `username` variable defined first, you will get a name error with th message `NameError: name 'username' is not defined`.
- **TypeError**: This is the error Python throws when you perform an operation on two or more incompatible data types. For example, you try to add a string to a number, you'll get the error `TypeError: can only concatenate str (not "int") to str`.
- **IndexError**: You'll get an `IndexError` if you access an index that does not exist in a list or other sequences like tuple and string. F example, in a `Hello world` string, the index of the last character is `11`. If you go ahead and access a character this way, `greet = "hello world"; print(greet[12])`, you'll get an error with the message `IndexError: string index out of range`.
- **AttributeError**: Python raises this error when you try to use a method or property that does not exist in an object of that type. For example, calling `.append()` on a string like `"hello".append("!")` will lead to an error with the message `AttributeError: 'str' object has no attribute 'append'`.

Good Debugging Techniques in Python

- **Using the `print` function**: Inserting `print()` statements around various points in your code while debugging helps you see the values of variables and how your code flows.
- **Using Python's Built-in Debugger (`pdb`)**: Python provides a `pdb` module for debugging. It's a part of the Python's standard librar so it's always available to use. With `pdb`, you can set a trace with the `set_trace()` method so you can start stepping through th code and inspect variables in an interactive way.
- **Leveraging IDE Debugging Tools**: Many integrated development environments (IDEs) and code editors like Pycharm and VS Code offer debugging tools with breakpoints, step execution, variable inspection, and other debugging features.

Exception Handling

- **`try...except`**: This is used to execute a block of code that might raise an exception. The `try` block is where you anticipate an error might occur, while the `except` block takes a specified exception and runs if that specified error is raised. Here's an example:

```
try:
  print(22 / 0)
except ZeroDivisionError:
  print('You can\'t divide by zero!')
  # You can't divide by zero!
```

  You can also chain multiple `except` blocks so you can handle more types of exceptions:

```
try:
  number = int(input('Enter a number: '))
  print(22 / number)
except ZeroDivisionError:
  print('You cannot divide by zero!')
  # You cannot divide by zero! prints when you enter 0
except ValueError:
  print('Please enter a valid number!')
  # Please enter a valid number! prints when you enter a string
```

- **`else`** and **`finally`**: These blocks extend `try...except`. If no exception occurs, the `else` block runs. The `finally` block always runs regardless of errors.

```
except ZeroDivisionError:
  print('You cannot divide by zero!') # This will not run
else:
  print(f'Result is {result}') # Result is 25.0
finally:
  print('Execution complete!') # Execution complete!
```

- **Exception Object**: This lets you access the exception itself for better debugging and printing the direct error message. To access th exception object, you need to use the `as` keyword. Here's an example:

```
try:
    value = int('This will raise an error')
except ValueError as e:
    print(f'Caught an error: {e}')
    # Caught an error: invalid literal for int() with base 10: 'This will raise an error'
```

- **The `raise` Statement**: This allows you to manually raise an exception. You can use it to throw an exception when a certain condition is met. Here's an example:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError('You cannot divide by zero')
    return a / b
```

Exception Signaling

The `raise` statement is also useful when you create your own custom exceptions, as you can use it to throw an exception with a custor message. Here's an example of that:

```
class InvalidCredentialsError(Exception):
    def __init__(self, message="Invalid username or password"):
        self.message = message
        super().__init__(self.message)

def login(username, password):
    stored_username = "admin"
    stored_password = "password123"

    if username != stored_username or password != stored_password:
        raise InvalidCredentialsError()

    return f"Welcome, {username}!"
```

Here's a how you can use the `login` function from the `InvalidCredentialsError` exception:

```
# failed login attempt
try:
    message = login("user", "wrongpassword")
    print(message)
except InvalidCredentialsError as e:
    print(f"Login failed: {e}")

# successful login attempt
try:
    # This block is not executed because the login was successful
    print(f"Login failed: {e}")
else:
    # The else block runs if the 'try' block completes without an exception
    print(message)
```

The `raise` statement can also be used with the `from` keyword to chain exceptions, showing the relationship between different errors

```
def parse_config(filename):
  try:
      with open(filename, 'r') as file:
          data = file.read()
          return int(data)
  except FileNotFoundError:
      raise ValueError('Configuration file is missing') from None
  except ValueError as e:
      raise ValueError('Invalid configuration format') from e

config = parse_config('config.txt')
```

Assignment

- [ ] Review the Error Handling topics and concepts.

Please complete the assignment

Submit

Ask for Help