Exception Handling: A Student Handout

Introduction to Exception Handling

In programming, an **exception** is an unexpected event that disrupts the normal flow of a program. Exception handling allows you to manage these situations gracefully, preventing program crashes.

Key Concept: What is an Exception?

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions. For example, dividing a number by zero raises an exception because it is mathematically undefined.

Built-in Exceptions

Python provides several built-in exceptions to handle common errors:

- **ZeroDivisionError**: Raised when dividing by zero.
- **ValueError**: Raised when a function receives an argument of the right type but an inappropriate value.
- **TypeError**: Raised when an operation is applied to an object of inappropriate type.
- **IndexError**: Raised when accessing an index that is out of range in a list or tuple.

Examples:

1. **ZeroDivisionError**:

```
try:
    result = 10 / 0
```

```
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

2. **ValueError**:

```
try:
    number = int("abc")
except ValueError:
    print("Invalid input; please enter a number.")
```

3. **IndexError**:

```
my_list = [1, 2, 3]
try:
    print(my_list[5])
except IndexError:
    print("Index out of range.")
```

Raising Exceptions

You can raise exceptions manually using the `raise` keyword to indicate an error condition.

Example:

1. **Raising a ValueError**:

```
def check_positive(number):
    if number < 0:
        raise ValueError("Negative numbers are not allowed!")
    return number

try:
    print(check_positive(-5))
except ValueError as e:
    print(e)
```

2. **Raising a TypeError**:

```
def add_numbers(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Both arguments must be integers.")
    return a + b

try:
    print(add_numbers(5, "10"))
except TypeError as e:
    print(e)
```

3. **Raising a Custom Exception**:

```
class CustomError(Exception):
    pass

def trigger_error():
    raise CustomError("This is a custom error.")

try:
    trigger_error()
except CustomError as e:
    print(e)
```

Handling Exceptions

Use a `try-except` block to handle exceptions and prevent program crashes.

Example:

1. **Handling ZeroDivisionError**:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Oops! You can't divide by zero.")
```

2. **Handling Multiple Exceptions**:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input; please enter a number.")
```

3. **Handling TypeError**:

```
try:
    result = "string" + 10
except TypeError:
    print("Cannot add a string and an integer.")
```

Finally Clause

The **finally clause** is a block of code that always executes, regardless of whether an exception occurred.

Example:

1. **File Handling**:

```
try:
    file = open("example.txt", "r")
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
    print("File closed.")
```

2. **Database Connection**:

```
try:
    connection = connect_to_database()
except ConnectionError:
    print("Failed to connect to database.")
```

```
    finally:
        connection.close()
        print("Connection closed.")
```

3. **Resource Cleanup**:

```
try:
    resource = acquire_resource()
except ResourceError:
    print("Resource acquisition failed.")
finally:
    release_resource(resource)
    print("Resource released.")
```

Activity: Write Code to Handle Exceptions

1. Write a program that asks the user to input two numbers.
2. Try to divide the first number by the second.
3. Handle the following exceptions:
    - **ZeroDivisionError**: If the user tries to divide by zero.
    - **ValueError**: If the user enters something that is not a number.

Sample Solution:

```
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    result = num1 / num2
    print(f"The result is: {result}")
except ZeroDivisionError:
    print("Error: You cannot divide by zero.")
except ValueError:
    print("Error: Please enter valid numbers.")
finally:
    print("Program execution completed.")
```

Conclusion

Exception handling is essential for writing robust programs. By using `try-except` blocks, raising exceptions, and utilizing the `finally` clause, you can manage unexpected situations effectively and prevent program crashes. Keep practicing to master exception handling!