**Python Control Flow and Error Handling Study Guide**

**1. Control Flow Statements: break, continue, and exit()**

In Python, break, continue, and exit() are used for different control flow operations. Here's a clear breakdown of their differences:

**1.1 break**

**Purpose**: Exits the **innermost loop** immediately.  
**Used in**: for or while loops.

**Example**:

for i in range(5):

if i == 3:

break

print(i)

# Output: 0 1 2

**1.2 continue**

**Purpose**: Skips the **rest of the current loop iteration** and moves to the next one.  
**Used in**: for or while loops.

**Example**:

for i in range(5):

if i == 3:

continue

print(i)

# Output: 0 1 2 4

**1.3 exit()**

**Purpose**: Terminates the **entire Python program**.  
**Used in**: Anywhere in the code (often in scripts or error handling).

**Example**:

import sys

for i in range(5):

if i == 3:

sys.exit()

print(i)

# Output: 0 1 2

# Program stops completely

**Note**: exit() and sys.exit() behave similarly; exit() is more common in interactive sessions, while sys.exit() is recommended in scripts.

**1.4 Summary Table**

| **Statement** | **Exits Loop** | **Skips to Next Iteration** | **Ends Program** |
| --- | --- | --- | --- |
| break | Yes | No | No |
| continue | No | Yes | No |
| exit() | No | No | Yes |

**2. Exception Handling with try-except**

In a try-except block, you can only catch actual exceptions—these are either built-in exceptions (like ValueError, TypeError, etc.) or custom exceptions that you define using Python classes.

**Important**: You cannot use except to catch arbitrary conditions like if x > 10 — those must be handled with regular if statements, not except.

**2.1 What Works**

**Catching a built-in exception**:

try:

x = int("abc")

except ValueError:

print("Caught a ValueError")

**Catching a custom exception**:

class MyCustomError(Exception):

pass

try:

raise MyCustomError("Something custom went wrong")

except MyCustomError:

print("Caught my custom error")

**2.2 What Does NOT Work**

This will raise a syntax error:

try:

x = 5

except x > 3: # Invalid

print("Can't do this")

**2.3 Handling Conditions via Exceptions**

If you want to handle a condition, do it like this:

x = 5

try:

if x > 3:

raise ValueError("x is too big!")

except ValueError:

print("Handled condition via exception")

**Key Point**: You can react to a self-defined condition, but only by raising an exception manually inside try, and then catching it in except.

**3. Assert Statement**

**3.1 What is Assert?**

✅ **Definition**: assert is a debugging aid, used to check if a condition is True. If the condition is False, Python raises an AssertionError.

**3.2 Syntax**

assert condition, "Optional error message"

**3.3 Example**

x = 5

assert x > 0, "x must be positive"

* If x > 0, nothing happens.
* If x <= 0, it raises: AssertionError: x must be positive

**3.4 Is Assert for Error Handling?**

❗ **Not really**. assert is mostly meant for debugging — checking internal conditions during development.

* ❌ **Don't use** assert for validating user input or production-level error handling.
* ✅ **Do use** assert to catch programming mistakes or bugs during development.

**3.5 Proper Error Handling Alternative**

Use try-except or raise specific exceptions:

x = -5

if x <= 0:

raise ValueError("x must be positive")

This is more appropriate for real error handling.

**3.6 When to Use Assert**

| **Situation** | **Use assert?** |
| --- | --- |
| Validating input from users | ❌ No |
| Checking developer assumptions | ✅ Yes |
| Preventing bugs during dev | ✅ Yes |
| Runtime error handling | ❌ No |

**3.7 Advanced Example**

def divide(a, b):

assert b != 0, "Division by zero!"

return a / b

print(divide(10, 2)) # OK

print(divide(10, 0)) # AssertionError

**4. Custom Exception Classes**

Creating your own exception classes is a clean and powerful way to handle errors in a structured way!

**4.1 Why Use Custom Error Handling with Classes?**

Creating your own exception classes helps you:

* Write more meaningful error messages
* Group related errors
* Catch specific error types in try-except blocks
* Keep your code clean and modular

**4.2 How to Create a Custom Exception Class**

**Step 1: Define the custom error class**

class MyCustomError(Exception):

pass

You're inheriting from Python's built-in Exception class.

**Step 2: Raise your custom exception**

x = -1

if x < 0:

raise MyCustomError("x must be non-negative")

**Step 3: Handle it with try-except**

try:

x = -1

if x < 0:

raise MyCustomError("x must be non-negative")

except MyCustomError as e:

print("Caught my custom error:", e)

**4.3 Adding More Logic to Your Custom Error**

You can make it smarter by customizing the constructor (\_\_init\_\_) and string representation (\_\_str\_\_):

class InputTooSmallError(Exception):

def \_\_init\_\_(self, value, minimum):

self.value = value

self.minimum = minimum

def \_\_str\_\_(self):

return f"Input {self.value} is too small. Minimum allowed is {self.minimum}."

**Usage**:

x = 3

if x < 5:

raise InputTooSmallError(x, 5)

**Output**: Input 3 is too small. Minimum allowed is 5.

**4.4 Best Practices**

| **Tip** | **Description** |
| --- | --- |
| Use meaningful names | Like InvalidUserInputError, CalculationError |
| Group exceptions | You can make all custom errors inherit from one base class |
| Keep messages informative | Use \_\_str\_\_() or constructor args for context |

**4.5 Complete Setup Example**

class CustomValidationError(Exception):

def \_\_init\_\_(self, field, msg):

self.field = field

self.msg = msg

def \_\_str\_\_(self):

return f"Validation failed for '{self.field}': {self.msg}"

def validate\_age(age):

if not isinstance(age, int):

raise CustomValidationError("age", "Must be an integer")

if age < 0:

raise CustomValidationError("age", "Cannot be negative")

return True

try:

validate\_age("twenty")

except CustomValidationError as e:

print("Error:", e)

**5. isinstance() Function**

**5.1 Purpose**

Check if a variable is an instance of a specific type (or a tuple of types).

**5.2 Syntax**

isinstance(object, type)

**Returns**: True or False

**5.3 Examples**

**Check if a variable is an int**:

x = 5

print(isinstance(x, int)) # True

**Check if a variable is a str**:

x = "hello"

print(isinstance(x, str)) # True

**Check if a variable is int or float**:

x = 3.14

print(isinstance(x, (int, float))) # True

**5.4 Why Use isinstance()?**

* **It's safe**: works correctly even with inheritance (unlike type(x) == int)
* **It's readable**: clearly shows what type check is being done
* **It's flexible**: you can check against multiple types at once

**5.5 Common Use Case**

def square(x):

if not isinstance(x, (int, float)):

raise TypeError("x must be a number")

return x \* x

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.  
A screenshot of a computer program

AI-generated content may be incorrect.  
A screenshot of a computer

AI-generated content may be incorrect.