Lecture 6 Handout

Exception Handling and Advanced List Operations

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 9 (Exception Handling), Chapter 7 (Advanced List Operations)
Reference Notebooks: `ch09/` (exception handling examples), `ch07/` (list operations examples)

Learning Objectives

By the end of this lecture, you will be able to:

1. **Understand** the notions of exceptions and runtime errors
2. **Use** the `try` statement to delimit code in which exceptions may occur
3. **Handle** exceptions with associated `except` clauses
4. **Use** the `try` statement's `else` clause to execute code when no exceptions occur
5. **Use** the `try` statement's `finally` clause to execute code regardless of whether an exception occurs
6. **Raise** exceptions to indicate runtime problems
7. **Understand** the traceback of functions and methods that led to an exception
8. **Master advanced list operations** including .append(), .extend(), .insert(), .remove(), .count(), .reverse(), and .copy()
9. **Sort and search lists efficiently** using .sort() method, sorted() function, .index() method, and membership testing
10. **Apply defensive programming principles** with both exception handling and advanced list operations

Prerequisites Review

Foundation Knowledge from Lectures 1-5:

From Lectures 1-2: Variables, data types, arithmetic operations, f-strings (Lecture 1); Complete if/elif/else mastery, boolean logic, string methods (Lecture 2)
From Enhanced Lectures 3-4: Lists, for/while loops, range() in all forms (Lecture 3); List comprehensions mastery - data processing and transformation (Lecture 4)
From Lecture 5: Function mastery - function definition, parameters, return values, scope (Lecture 5)

Transformation Goal: Move from function-oriented programming to **defensive programming with exception handling** - building robust applications that handle errors gracefully rather than crashing.

1 Part I: Understanding Exceptions and Runtime Errors

1.1 What Are Exceptions?

Think of exceptions as Python's way of saying "Hold on, something's wrong!" instead of just crashing your program. Just like a car's dashboard warning lights tell you about problems, exceptions tell you exactly what went wrong so you can fix it.

Definition: Exceptions are Python's alert system that signals when something unexpected happens during program execution, giving you the chance to handle the problem gracefully rather than letting your program crash.

Here are the most common exceptions you'll encounter when working with user input and data:

Common Exception Types (with everyday examples):

- **ValueError**: Like trying to use "hello" as a number - `int("hello")` makes no sense!
- **ZeroDivisionError**: Dividing by zero is mathematically impossible - your calculator would show "ERROR"
- **IndexError**: Asking for the 10th item in a 3-item list - there's nothing there!
- **FileNotFoundError**: Looking for a file that doesn't exist - like searching for a missing book
- **AttributeError**: Using a method that doesn't exist - like trying to `.append()` to a string

Practice Tip: Try each of these in your Python interpreter to see what the error messages look like!

1.2 Seeing Exceptions in Action

Division By Zero Example: Think about your calculator - what happens when you try to divide by zero? It shows an error, right? Python does the same thing:

```
# This will crash your program with a ZeroDivisionError
result = 10 / 0
print("This line never runs because the program crashed above")
```

Invalid Input Example: Imagine asking someone "What's your age?" and they answer "twenty-five". You can't do math with words! Same with Python:

```python
# This will crash if user types "hello" instead of a number
age = int(input('Enter your age: '))
print(f"Next year you'll be {age + 1}")
```

The Problem: Both examples crash your program completely. Users hate crashed programs! We need a better way...

2 Part II: The try Statement - Your Safety Net

2.1 Basic Exception Handling Structure

Here's the solution: instead of letting exceptions crash your program, we can **catch** them and handle them gracefully. Think of it like wearing a seatbelt - you hope nothing goes wrong, but if it does, you're protected!

The `try/except` structure is like saying: "Try to do this risky thing, but if something goes wrong, I have a backup plan."

Real-World Example: Smart Division Calculator

Let's build a division calculator that doesn't crash when users make mistakes:

```python
# dividebyzero.py
"""A robust division calculator that handles user errors gracefully."""

print("Welcome to the Smart Division Calculator!")
print("I can handle your mistakes gracefully - try me!\n")

while True:
    try:
        # The "risky" part - user input and division
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2

    except ValueError:  # User typed words instead of numbers
        print('Oops! Please enter actual numbers, not words.\n')

    except ZeroDivisionError:  # User tried to divide by zero
        print('Nice try, but division by zero breaks mathematics!\n')

    else:  # Only runs if nothing went wrong
        print(f'Success! {number1} / {number2} = {result:.3f}')
        break  # Exit the loop when we succeed

print("Calculator finished. Thanks for using the smart version!")
```

Why This Works Better:

- Program never crashes - it just gives helpful messages
- Users can try again instead of having to restart
- Professional software behavior that users expect

2.2 Understanding the Components (The Anatomy)

Think of exception handling like a restaurant's food safety system:

try Clause (The Kitchen):

- This is where the "risky cooking" happens
- Put any code that might fail inside the `try` block
- Like a chef trying a new recipe - it might work perfectly, or it might not

except Clause (The Backup Plan):

- These are your backup plans for specific problems
- Each `except` handles one type of problem (like different kitchen emergencies)
- `except ValueError` = "What to do if numbers are invalid"
- `except ZeroDivisionError` = "What to do if someone tries to divide by zero"

else Clause (The Success Celebration):

- Only runs when everything in `try` worked perfectly
- Like saying "Great! Nothing went wrong, let's proceed with the success case"
- Perfect place for code that should only run when no errors occurred

Simple Pattern: "Try this risky thing, but if X goes wrong, do Y instead."

2.3 Flow of Control for Exception Handling

When a ZeroDivisionError occurs:

- The point where an exception occurs is the **raise point**
- When an exception occurs in a `try` suite, it terminates immediately
- Program control transfers to the matching `except` handler
- After handling, execution resumes after the `try` statement

When no exceptions occur:

- Program execution resumes with the `else` clause (if present)
- Otherwise, execution continues with the next statement after the `try` statement

3 Part III: Catching Multiple Exceptions

3.1 Multiple Exception Types in One Handler

If several `except` suites are identical, you can catch those exception types by specifying them as a tuple in a *single* `except` handler:

```
try:
    num = int(input("Please enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"An error occurred: {e}")
```

Key Points:

- `as` clause is optional - typically programs don't need to reference the exception object directly
- Can use the variable in the `as` clause to reference the exception object in the `except` suite

3.2 Exception Handling Guidelines

What Code Should Be Placed in a try Suite?

- Each `try` statement should enclose a section of code small enough that when an exception occurs, the specific context is known
- The `except` handlers can process the exception properly
- If many statements raise the same exception types, multiple `try` statements may be required

4 Part IV: The finally Clause

4.1 Understanding the finally Clause

The `try` statement may have a `finally` clause after any `except` clauses or the `else` clause.

Key Characteristics:

- **finally** clause is guaranteed to execute
- Executes regardless of whether exceptions occur
- In other languages, ideal for resource-deallocation code
- In Python, we prefer the `with` statement for resource management

4.2 Example of finally Clause

```
try:
    print('try suite with no exceptions raised')
except:
    print('this will not execute')
else:
    print('else executes because no exceptions in the try suite')
finally:
    print('finally always executes')
```

4.3 Combining with and try...except Statements

Most resources requiring explicit release have potential exceptions. Robust file-processing code normally appears in a `try` suite containing a `with` statement:

```
try:
    with open('gradez.txt', 'r') as accounts:
        print(f'{"ID":<3}{"Name":<7}{"Grade"}')
        for record in accounts:
            student_id, name, grade = record.split()
            print(f'{student_id:<3}{name:<7}{grade}')
except FileNotFoundError:
    print('The file name you specified does not exist')
```

5 Part V: Explicitly Raising an Exception

5.1 Using the raise Statement

Sometimes you need to write functions that raise exceptions to inform callers of errors.

The **raise** statement explicitly raises an exception:

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError("Insufficient funds")
    balance -= amount
    return balance

try:
    balance = 100
    amount = 150
    new_balance = withdraw(balance, amount)
    print(f"New balance: {new_balance}")
except ValueError as e:
    print(f"Error: {e}")
```

Key Points:

- Creates an object of the specified exception class
- Exception class name may be followed by parentheses containing arguments
- Typically includes a custom error message string
- It's recommended to use Python's built-in exception types

5.2 Custom Exception Classes

You can also create custom exception classes:

```
class InsufficientFundsError(Exception):
    pass

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError("Insufficient funds in your
            account")
    balance -= amount
    return balance

try:
    withdraw(100, 150)
except InsufficientFundsError as e:
    print(f"Error: {e}")
```

6 Part VI: Stack Unwinding and Tracebacks - Your Debugging Superpower

6.1 Understanding Tracebacks (Python's Crime Scene Investigation)

When your program crashes, Python doesn't just give up - it leaves you a detailed "crime scene report" called a traceback. This tells you exactly what happened and where, like a detective's investigation notes.

Think of your program like a stack of function calls, like layers in a cake. When something goes wrong, Python shows you every layer that led to the problem:

```
def process_student_grade():
    """Main function that processes student data."""
    print("Starting grade processing...")
    calculate_final_grade()

def calculate_final_grade():
    """Calculate the final grade for a student."""
    print("Calculating final grade...")
    check_homework_score()

def check_homework_score():
    """Check if homework score is valid."""
    print("Checking homework...")
    score = int("invalid_number")  # This will crash!

# When you call this, you get a detailed traceback:
process_student_grade()
```

6.2 Reading Tracebacks Like a Detective

When the above code crashes, Python shows you this "investigation report":

How to Read the Report (Bottom to Top):

- **Step 1**: Read the error message at the bottom first - "What went wrong?"
- **Step 2**: Look at the line that actually failed - "Where did it break?"
- **Step 3**: Trace backwards up the call stack - "How did we get here?"
- **Step 4**: Find YOUR code (not library code) - "What can I fix?"

Professional Debugging Workflow:

1. Read the error message: "ValueError: invalid literal for int()"
2. Find the problem line: `score = int("invalid_number")`
3. Trace the path: `process_student_grade → calculate_final_grade → check_homework_score`
4. Fix the root cause: Validate the input before converting

Practice Tip: Don't be intimidated by long tracebacks! They're your friends - they tell you exactly what to fix.

7 Key Concepts Summary

Complete Programming Foundation Mastery Achieved:

Core Technical Skills:

1. **Exception Recognition and Handling** - Understanding runtime errors, using try/except/else/finally structure
2. **Multiple Exception Handling** - Catching different exception types and safe error management

3. **Resource Management** - Combining exception handling with file operations and cleanup
4. **Raising Exceptions** - Creating and throwing custom exceptions for robust applications
5. **Debugging with Tracebacks** - Understanding error information and call stacks
6. **Advanced List Operations** - Dynamic list building with .append(), .extend(), .insert()
7. **List Data Management** - Removing elements with .remove(), .clear(), analyzing with .count()
8. **List Sorting and Searching** - Efficient sorting with .sort(), sorted(), searching with .index(), membership testing

Professional Development Impact:
Robust Programming Benefits:

- Handle errors gracefully instead of crashing
- Provide helpful error messages to users
- Maintain program stability when problems occur
- Create reliable, user-friendly software

Professional Exception Handling Rules (Your Programming Standards):

- **Be Specific**: Catch `ValueError`, not just `Exception` - you want to know exactly what went wrong
- **Help Your Users**: Write error messages that normal people can understand ("Please enter a number" not "invalid literal for int()")
- **Clean Up After Yourself**: Use `finally` for closing files or connections - like washing dishes after cooking
- **Validate Early**: Check input before processing - like checking ingredients before cooking
- **Master Traceback Reading**: This skill separates junior from senior developers!

Real-World Application Checklist:

1. Does your program crash when users type the wrong thing? → Add exception handling
2. Do your error messages help users fix their mistakes? → Improve the messages
3. Can you read and understand Python tracebacks quickly? → Practice this skill!
4. Do you validate input before processing it? → Add validation functions

8 Part VII: Advanced List Operations

8.1 Introduction to Advanced List Management

Now that you've mastered exception handling for robust programming, we'll expand your data structure capabilities with advanced list operations. These powerful methods build on the basic list operations from Lectures 3-4, adding professional-grade functionality that makes lists versatile for complex applications.

Think of advanced list methods like specialized tools in a workshop - each method is designed for specific tasks and provides the most efficient approach for different data manipulation needs. Understanding when to use each method is crucial for professional programming: adding single items versus multiple items, removing by value versus by position, and maintaining versus changing list order.

8.2 Building Dynamic Lists with .append() and .extend()

The fundamental difference between these methods is crucial for professional programming:

```
# Building dynamic student grade lists with .append() method
print("=== Advanced List Operations Demo ===")

# Start with empty list and build dynamically
student_grades = []  # Empty list for building
print(f"Initial list: {student_grades}")

# Adding individual grades (.append() for single items)
student_grades.append(95)  # Add first grade
student_grades.append(87)  # Add second grade
student_grades.append(92)  # Add third grade
print(f"After adding grades: {student_grades}")
```

Key Point: The .append() method adds the entire argument as a single item to the end of the list, making it perfect for building lists one element at a time.

8.3 Understanding .insert() for Positional Control

The .insert() method gives you complete control over where new items are placed:

```
# Demonstrating .insert() method for positional control
color_names = ['orange', 'yellow', 'green']
print(f"Original colors: {color_names}")

# Insert at specific position
color_names.insert(0, 'red')  # Insert at beginning
print(f"After inserting 'red' at position 0: {color_names}")

# Adding to the end with .append()
color_names.append('blue')
print(f"After appending 'blue': {color_names}")
```

Unlike .append() which always adds to the end, .insert() lets you specify exactly where the new item should go.

8.4 Adding Multiple Elements with .extend()

The .extend() method adds each individual item from another iterable to your list:

```
# Using .extend() to add multiple items efficiently
color_names.extend(['indigo', 'violet'])
print(f"After extending with ['indigo', 'violet']: {color_names}")

# Show extend works with different sequence types
sample_list = []
s = 'abc'
sample_list.extend(s)  # Extend with string
print(f"After extending with string 'abc': {sample_list}")

t = (1, 2, 3)
sample_list.extend(t)  # Extend with tuple
print(f"After extending with tuple (1, 2, 3): {sample_list}")
```

Critical Understanding: .extend() adds each individual item from the iterable, while .append() would add the entire list as a single item.

8.5 Removing Elements: .remove(), .clear(), and More

Various methods provide different approaches to element removal:

```
# Removing the first occurrence of an element
color_names.remove('green')  # Remove first occurrence of 'green'
print(f"After removing 'green': {color_names}")

# Emptying a list with .clear()
test_colors = color_names.copy()
print(f"Before clear: {test_colors}")
test_colors.clear()
print(f"After clear: {test_colors}")
```

8.6 Counting and Analyzing List Contents

The .count() method helps analyze data frequency:

```
# Demonstrating .count() method with survey responses
responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3,
             1, 4, 3, 3, 3, 2, 3, 3, 2, 2]

# Count occurrences of each response value
for i in range(1, 6):
    count = responses.count(i)
    print(f'{i} appears {count} times in responses')
```

8.7 List Manipulation: .reverse() and .copy()

Additional useful list methods for data manipulation:

```
# Reversing a list's elements in place
color_names = ['red', 'orange', 'yellow', 'green', 'blue']
print(f"Original order: {color_names}")
color_names.reverse()
print(f"After reverse: {color_names}")

# Creating independent copies
original_list = ['a', 'b', 'c']
copied_list = original_list.copy()
print(f"Original: {original_list}")
print(f"Copy: {copied_list}")

# Modify copy to show they're independent
copied_list.append('d')
print(f"After modifying copy: original={original_list}, copy={
    copied_list}")
```

9 Part VIII: List Sorting and Searching

9.1 Sorting Lists in Place and Creating Sorted Copies

Python provides two approaches to sorting: the .sort() method modifies a list in place, while the sorted() function returns a new sorted list without modifying the original.

```python
# Sorting demonstration with .sort() method
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
print(f"Original numbers: {numbers}")

# Sort in ascending order (modifies original)
numbers.sort()
print(f"After .sort(): {numbers}")

# Sort in descending order
numbers.sort(reverse=True)
print(f"After .sort(reverse=True): {numbers}")
```

9.2 Using sorted() Function

The sorted() function creates new sorted sequences without modifying originals:

```python
# Using sorted() function (creates new list)
original_numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
print(f"Original: {original_numbers}")

# Create sorted copy
ascending_numbers = sorted(original_numbers)
print(f"Sorted copy: {ascending_numbers}")
print(f"Original unchanged: {original_numbers}")

# sorted() works with any sequence type
letters = 'fadgchjebi'
ascending_letters = sorted(letters)
print(f"Original string: {letters}")
print(f"Sorted letters: {ascending_letters}")
```

9.3 Searching Sequences with index() Method

Searching locates particular values in sequences using various methods:

```python
# Using list.index() method for searching
numbers = [3, 7, 1, 5, 2, 8, 5, 6]
print(f"Search list: {numbers}")

# Find index of first occurrence
first_five_index = numbers.index(5)
print(f"First occurrence of 5 at index: {first_five_index}")

# Search starting from specific index
numbers_doubled = numbers * 2  # Create list with duplicates
print(f"Doubled list: {numbers_doubled}")
second_five_index = numbers_doubled.index(5, 7)  # Start search from
    index 7
print(f"Next occurrence of 5 starting from index 7: {second_five_index}
    ")
```

9.4 Membership Testing with in and not in

Use membership operators for safe value checking:

```python
# Using in and not in operators for membership testing
numbers = [3, 7, 1, 4, 2, 8, 5, 6]

# Test membership with if statement
if 5 in numbers:
    print('5 is in the list')

# Direct boolean results
print(f"5 in numbers: {5 in numbers}")
print(f"1000 not in numbers: {1000 not in numbers}")

# Safe searching to prevent ValueError
search_key = 1000
if search_key in numbers:
    print(f'Found {search_key} at index {numbers.index(search_key)}')
else:
    print(f'{search_key} not found')
```

Connection to Next Lecture

Your Programming Evolution:

1. **Lectures 1-2**: Variables, operations, and basic control structures
2. **Lectures 3-4**: Basic lists, loops, and data processing with comprehensions
3. **Lecture 5**: Function-oriented programming with modular design
4. **Lecture 6**: Defensive programming with exception handling + Advanced list operations

Exception handling is your gateway from writing "student programs" to building professional software that users actually want to use. The defensive programming principles you've mastered will serve as the foundation for all your future programming projects.

Your Next Steps (Homework Challenge):

1. Take any program you've written before and add proper exception handling
2. Practice reading tracebacks - break some code on purpose and fix it using the traceback
3. Build a small program that asks for user input and handles ALL possible errors gracefully
4. Challenge: Create a program that NEVER crashes, no matter what the user types

---

Professional Mindset: Exception handling separates amateur code from professional software. Users should never see your program crash - they should see helpful messages that guide them to success. Master this skill, and you're thinking like a professional developer!