Lecture 5 Handout

Functions & Modules

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 4, Functions
Reference Notebooks: `ch04/04_02.ipynb` (function definition), `ch04/04_03.ipynb` (multiple parameters), `ch04/04_04.ipynb` (random module), `ch04/04_07.ipynb` (math module)

Prerequisites Review

Building on Your Enhanced Knowledge Foundation:

From Lectures 1-2: Complete mastery of variables, data types, arithmetic operations, input/output with f-strings, decision structures (if/elif/else), boolean logic, string methods (.isdigit(), .lower(), .upper(), .strip())
From Enhanced Lecture 3: List fundamentals (creation, indexing, slicing), for loops with range() in all forms, while loops, nested control structures
From Lecture 4: List comprehensions mastery - data processing, mapping, filtering, elegant syntax [expression for item in iterable if condition]

Transformation Goal: This lecture transforms you from **procedural programming with comprehensions** to **function-oriented modular programming** - organizing code into reusable, maintainable components with enhanced capabilities through modules.

Learning Objectives

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

1. **Master function definition** using def keyword with proper syntax, parameters, and return statements
2. **Design functions with multiple parameters** for flexible data processing and calculations
3. **Implement return values effectively** including single values and tuple unpacking for multiple returns
4. **Import and utilize Python Standard Library modules** (random, math) for enhanced functionality

5. **Apply scope rules correctly** to understand variable accessibility and namespace concepts
6. **Generate random numbers** for simulations, games, and data generation using random module
7. **Perform mathematical calculations** using math module functions and constants
8. **Organize code modularly** using function-based architecture for maintainable programs
9. **Integrate functions with previous knowledge** (comprehensions, loops, control structures)

1 Today's Learning Journey: From Procedural to Functional Programming

This lecture introduces the fundamental concept of **functions** - reusable blocks of code that perform specific tasks, and **modules** - Python's system for organizing and sharing functionality. We'll transform your procedural programming approach into function-oriented design that emphasizes modularity, reusability, and enhanced capabilities.

Why Functions Matter: The Foundation of Modular Programming

Functions solve critical programming challenges:

- **Code Reusability:** Write once, use many times - eliminates repetitive coding
- **Modularity:** Break complex problems into manageable, testable components
- **Abstraction:** Hide implementation details while providing clear interfaces
- **Maintainability:** Changes in one place automatically propagate throughout program
- **Organization:** Structure programs logically with clear separation of concerns

2 Part I: Function Fundamentals (25 minutes)

2.1 Function Definition and Basic Syntax

Definition: Functions are reusable blocks of code that perform specific tasks and can accept input through parameters and return output.

Function Anatomy:

```
def function_name(parameters):
    """Optional docstring"""
        # Function body
        return result
```

Example: Simple square function (building on arithmetic operations from Lecture 1)

```
def square(number):
    """Calculate the square of number."""
    return number ** 2

# Function calls
result1 = square(7)       # Returns 49
```

```
result2 = square(2.5)     # Returns 6.25
print(f"Square of 7 is {result1}")
print(f"Square of 2.5 is {result2}")
```

2.2 Function Components Explained

1. Function Header:

- `def` keyword begins function definition
- Function name follows Python naming conventions (lowercase, underscores)
- Parameters in parentheses specify inputs
- Colon (:) ends the header

2. **Docstring:** First line describes function purpose (best practice)
3. **Function Body:** Indented code block performing the task
4. **Return Statement:** Sends result back to caller (optional)

2.3 Parameter Passing and Data Flow

Definition: Parameters allow functions to receive data from outside, arguments are the actual values passed to parameters.

Example: Calculator functions building on mathematical operations

```
def calculate_area(length, width):
    """Calculate rectangular area from length and width."""
    area = length * width
    return area

# Data flow demonstration
room_length = 12.5
room_width = 10.0
room_area = calculate_area(room_length, room_width)
print(f"Room area: {room_area} square feet")

# Direct function calls
garden_area = calculate_area(8, 6)  # Arguments: 8, 6
print(f"Garden area: {garden_area} square feet")
```

2.4 Return Values and Function Output

Three ways functions can return values:

```
# 1. Return with expression
def add_numbers(a, b):
    """Return sum of two numbers."""
    return a + b

# 2. Return without expression (returns None)
def print_greeting(name):
    """Print greeting message."""
    print(f"Hello, {name}!")
    return  # Explicitly returns None

# 3. No return statement (implicitly returns None)
```

```python
def display_info():
    """Display program information."""
    print("Welcome to Python Functions!")

# Usage examples
sum_result = add_numbers(15, 25)      # Returns 40
greeting_result = print_greeting("Alice")  # Returns None
info_result = display_info()          # Returns None

print(f"Sum: {sum_result}")
print(f"Greeting result: {greeting_result}")
print(f"Info result: {info_result}")
```

3 Part II: Function Parameters and Applications (20 minutes)

3.1 Multiple Parameter Functions

Definition: Functions can accept multiple parameters for flexible data processing and complex calculations.

Example: Maximum function with three parameters

```python
def maximum(value1, value2, value3):
    """Return the maximum of three values."""
    max_value = value1
    if value2 > max_value:
        max_value = value2
    if value3 > max_value:
        max_value = value3
    return max_value

# Testing with different data types
print(maximum(12, 27, 36))            # 36
print(maximum(12.3, 45.6, 9.7))      # 45.6
print(maximum('yellow', 'red', 'orange'))  # 'yellow'
print(maximum(13.5, -3, 7))          # 13.5
```

Note: Python's built-in `max()` and `min()` functions provide this functionality, demonstrating the principle of using existing solutions rather than reinventing.

3.2 Functions with Data Processing (Building on List Comprehensions)

Integration Strategy: Combine functions with list comprehensions from Lecture 4 for powerful data processing.

```python
def process_numbers(numbers):
    """Process a list of numbers using comprehensions."""
    # Filter positive numbers and square them
    positive_squares = [num ** 2 for num in numbers if num > 0]
    return positive_squares

def calculate_statistics(data):
    """Calculate basic statistics for a list of numbers."""
    if not data:  # Handle empty list
        return 0, 0, 0  # Return tuple: count, total, average

    count = len(data)
    total = sum(data)
```

```
    average = total / count

    return count, total, average  # Multiple return values

# Usage with comprehensions
test_numbers = [-3, 5, -1, 8, 2, -7, 4]
processed = process_numbers(test_numbers)
print(f"Positive squares: {processed}")

# Statistics calculation
stats_count, stats_total, stats_average = calculate_statistics(
    processed)
print(f"Count: {stats_count}, Total: {stats_total}, Average: {
    stats_average:.2f}")
```

3.3 Validation Functions (Building on String Methods)

Integration with Previous Knowledge: Use string methods from Lectures 1-2 within functions for robust input validation.

```
def validate_age(age_input):
    """Validate age input using string methods."""
    # Clean input
    cleaned_age = age_input.strip()

    # Check if it's a valid number
    if not cleaned_age.isdigit():
        return False, "Age must be a positive number"

    # Convert and check range
    age = int(cleaned_age)
    if age < 0 or age > 150:
        return False, "Age must be between 0 and 150"

    return True, age

def validate_name(name_input):
    """Validate name input using string methods."""
    # Clean and format name
    cleaned_name = name_input.strip().title()

    # Check if name contains only alphabetic characters and spaces
    if not all(char.isalpha() or char.isspace() for char in
        cleaned_name):
        return False, "Name must contain only letters and spaces"

    if len(cleaned_name) < 2:
        return False, "Name must be at least 2 characters long"

    return True, cleaned_name

# Usage examples
age_valid, age_result = validate_age("25")
name_valid, name_result = validate_name("  john doe  ")

print(f"Age validation: {age_valid}, Result: {age_result}")
print(f"Name validation: {name_valid}, Result: {name_result}")
```

3.4 Scope Rules and Variable Accessibility

Definition: Scope determines where variables can be accessed - local variables exist only within functions, global variables exist throughout the program.

Scope Visualization:

Global Scope (accessible everywhere)
`global_var = 100`

Function A
`local_a = 10`
`param_a`
(can access global_var)

Function B
`local_b = 20`
`param_b`
(can access global_var)

Note: Local variables cannot access each other

```python
# Global variable
account_balance = 1000.0

def deposit(amount):
    """Add money to account (local scope demonstration)."""
    # Local variable
    transaction_fee = 2.00
    net_deposit = amount - transaction_fee

    # Access global variable (read-only without global keyword)
    new_balance = account_balance + net_deposit

    print(f"Depositing ${amount:.2f}")
    print(f"Transaction fee: ${transaction_fee:.2f}")
    print(f"Net deposit: ${net_deposit:.2f}")
    print(f"New balance would be: ${new_balance:.2f}")

    return new_balance

def get_account_info():
    """Display account information."""
    print(f"Current balance: ${account_balance:.2f}")

# Function calls
get_account_info()
result_balance = deposit(50.00)

# These would cause NameError - local variables not accessible
# print(transaction_fee)  # Error!
# print(net_deposit)      # Error!
```

4 Part III: Modules and Standard Library (20 minutes)

4.1 Module System Introduction

Definition: Modules are files containing Python code that extend language functionality through imports. They organize related functions and avoid code duplication.

Import Syntax and Usage:

```python
# Method 1: Import entire module
import random
```

```
import math

# Method 2: Import specific functions (advanced - use with caution)
# from random import randint, choice

# Method 3: Import with alias (advanced)
# import random as rnd
```

4.2 Random Module: Games and Simulations

Key Functions:

- `random.randrange(start, stop)` - Integer in range [start, stop)
- `random.randint(a, b)` - Integer in range [a, b] (inclusive)
- `random.choice(sequence)` - Random element from sequence
- `random.random()` - Float between 0.0 and 1.0
- `random.seed(value)` - Set seed for reproducibility

Example: Dice Games and Simulations

```
import random

def roll_die(sides=6):
    """Roll a die with specified number of sides."""
    return random.randrange(1, sides + 1)

def dice_game():
    """Simple dice game demonstrating random numbers."""
    print("Welcome to the Dice Game!")

    # Player rolls
    player_roll = roll_die()
    computer_roll = roll_die()

    print(f"You rolled: {player_roll}")
    print(f"Computer rolled: {computer_roll}")

    if player_roll > computer_roll:
        return "You win!"
    elif computer_roll > player_roll:
        return "Computer wins!"
    else:
        return "It's a tie!"

def random_password_generator(length=8):
    """Generate random password using random.choice."""
    characters = "
       abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    password = ""

    for i in range(length):
        password += random.choice(characters)

    return password

```

```python
def simulate_coin_flips(num_flips):
    """Simulate coin flips and count results."""
    heads_count = 0
    tails_count = 0

    for flip in range(num_flips):
        # 0 = tails, 1 = heads
        if random.randrange(2) == 1:
            heads_count += 1
        else:
            tails_count += 1

    return heads_count, tails_count

# Usage demonstrations
print(dice_game())
print(f"Random password: {random_password_generator(12)}")

heads, tails = simulate_coin_flips(1000)
print(f"In 1000 flips: {heads} heads, {tails} tails")
```

Seeding for Reproducibility:

```python
# Set seed for consistent results (useful for testing)
random.seed(42)
print("First sequence:")
for i in range(5):
    print(random.randrange(1, 7), end=" ")

print("\nSecond sequence (different):")
for i in range(5):
    print(random.randrange(1, 7), end=" ")

# Reset same seed for identical results
random.seed(42)
print("\nThird sequence (same as first):")
for i in range(5):
    print(random.randrange(1, 7), end=" ")
```

4.3 Math Module: Mathematical Functions and Constants

Key Functions and Constants:

- `math.sqrt(x)` - Square root
- `math.pow(x, y)` - x raised to power y
- `math.pi` - Mathematical constant π (3.14159...)
- `math.e` - Mathematical constant e (2.71828...)
- `math.sin()`, `math.cos()`, `math.tan()` - Trigonometric functions
- `math.ceil()`, `math.floor()` - Ceiling and floor functions

Example: Geometric Calculator Functions

```python
import math

def circle_area(radius):
    """Calculate area of circle using math.pi."""
    return math.pi * radius ** 2

def circle_circumference(radius):
    """Calculate circumference of circle."""
    return 2 * math.pi * radius

def distance_between_points(x1, y1, x2, y2):
    """Calculate distance between two points using math.sqrt."""
    dx = x2 - x1
    dy = y2 - y1
    distance = math.sqrt(dx ** 2 + dy ** 2)
    return distance

def triangle_hypotenuse(a, b):
    """Calculate hypotenuse using Pythagorean theorem."""
    return math.sqrt(a ** 2 + b ** 2)

def compound_interest(principal, rate, time, compounds_per_year):
    """Calculate compound interest using math.pow."""
    # A = P(1 + r/n)^(nt)
    amount = principal * math.pow(1 + rate/compounds_per_year,
        compounds_per_year * time)
    return amount

# Usage examples
radius = 5.0
print(f"Circle with radius {radius}:")
print(f"Area: {circle_area(radius):.2f}")
print(f"Circumference: {circle_circumference(radius):.2f}")

print(f"\nDistance from (0,0) to (3,4): {distance_between_points(0, 0,
    3, 4):.2f}")

print(f"Right triangle with sides 3 and 4:")
print(f"Hypotenuse: {triangle_hypotenuse(3, 4):.2f}")

investment = compound_interest(1000, 0.05, 10, 12)
print(f"\n$1000 at 5% for 10 years (monthly compounding): ${investment
    :.2f}")
```

Mathematical Constants and Advanced Functions:

```python
# Constants demonstration
print(f"Pi: {math.pi:.6f}")
print(f"e: {math.e:.6f}")

# Rounding functions
value = 4.7
print(f"Floor of {value}: {math.floor(value)}")  # 4
print(f"Ceiling of {value}: {math.ceil(value)}")  # 5

# Logarithmic functions
print(f"Natural log of e: {math.log(math.e):.2f}")  # 1.00
print(f"Log base 10 of 100: {math.log10(100):.1f}")  # 2.0
```

```
13
14 # Trigonometric functions (angles in radians)
15 angle_degrees = 45
16 angle_radians = math.radians(angle_degrees)  # Convert to radians
17 print(f"sin({angle_degrees} degrees): {math.sin(angle_radians):.6f}")
18 print(f"cos({angle_degrees} degrees): {math.cos(angle_radians):.6f}")
```

5 Part IV: Integration and Applications (10 minutes)

5.1 Complete Program Integration

Bringing It All Together: Functions + Modules + Previous Knowledge

Example: Student Grade Management System

```
1  import random
2  import math
3
4  def generate_test_scores(num_students, min_score=60, max_score=100):
5      """Generate random test scores for simulation."""
6      scores = []
7      for i in range(num_students):
8          score = random.randrange(min_score, max_score + 1)
9          scores.append(score)
10     return scores
11
12 def calculate_grade_statistics(scores):
13     """Calculate comprehensive statistics using math functions."""
14     if not scores:
15         return None
16
17     count = len(scores)
18     total = sum(scores)
19     average = total / count
20
21     # Calculate standard deviation
22     variance_sum = sum((score - average) ** 2 for score in scores)
23     variance = variance_sum / count
24     std_deviation = math.sqrt(variance)
25
26     # Find min and max
27     min_score = min(scores)
28     max_score = max(scores)
29
30     return {
31         'count': count,
32         'average': average,
33         'std_deviation': std_deviation,
34         'min_score': min_score,
35         'max_score': max_score
36     }
37
38 def assign_letter_grades(scores):
39     """Assign letter grades based on scores."""
40     # Using list comprehension with function
41     def score_to_grade(score):
42         if score >= 90:
43             return 'A'
```

```
        elif score >= 80:
            return 'B'
        elif score >= 70:
            return 'C'
        elif score >= 60:
            return 'D'
        else:
            return 'F'

    # Comprehension with function call
    grades = [score_to_grade(score) for score in scores]
    return grades

def analyze_class_performance(num_students=30):
    """Complete class performance analysis."""
    print(f"Analyzing performance for {num_students} students...")

    # Generate data
    scores = generate_test_scores(num_students)
    grades = assign_letter_grades(scores)
    stats = calculate_grade_statistics(scores)

    # Display results
    print(f"\nClass Statistics:")
    print(f"Average: {stats['average']:.1f}")
    print(f"Standard Deviation: {stats['std_deviation']:.1f}")
    print(f"Range: {stats['min_score']} - {stats['max_score']}")

    # Grade distribution using comprehensions
    grade_counts = {
        'A': len([g for g in grades if g == 'A']),
        'B': len([g for g in grades if g == 'B']),
        'C': len([g for g in grades if g == 'C']),
        'D': len([g for g in grades if g == 'D']),
        'F': len([g for g in grades if g == 'F'])
    }

    print(f"\nGrade Distribution:")
    for grade, count in grade_counts.items():
        percentage = (count / num_students) * 100
        print(f"{grade}: {count} students ({percentage:.1f}%)")

    return scores, grades, stats

# Run the analysis
scores, grades, statistics = analyze_class_performance(25)
```

5.2 Function-Based Program Architecture

Main Function Pattern: Organize programs with a main() function

```
def main():
    """Main program function - entry point."""
    print("Welcome to the Python Functions Demo!")

    while True:
        print("\nChoose an option:")
        print("1. Play dice game")
```

```
        print("2. Generate password")
        print("3. Calculate circle area")
        print("4. Analyze class performance")
        print("5. Exit")

        choice = input("Enter choice (1-5): ").strip()

        if choice == '1':
            result = dice_game()
            print(result)
        elif choice == '2':
            length = int(input("Password length: "))
            password = random_password_generator(length)
            print(f"Generated password: {password}")
        elif choice == '3':
            radius = float(input("Enter radius: "))
            area = circle_area(radius)
            print(f"Circle area: {area:.2f}")
        elif choice == '4':
            num_students = int(input("Number of students: "))
            analyze_class_performance(num_students)
        elif choice == '5':
            print("Thank you for using the program!")
            break
        else:
            print("Invalid choice. Please try again.")

# Program entry point
if __name__ == "__main__":
    main()
```

Key Concepts Summary

Functions:

- Defined with `def` keyword, parameters in parentheses
- Can return single values, multiple values (tuples), or None
- Parameters exist only during function execution (local scope)
- Enable code reusability, modularity, and abstraction

Modules:

- `random` module: randrange(), randint(), choice(), seed()
- `math` module: sqrt(), pow(), pi, trigonometric functions
- Import with `import module_name` syntax
- Access functions with dot notation: `module.function()`

Integration Patterns:

- Functions using list comprehensions for data processing
- Functions with control structures for complex logic
- Function-based program organization with main() pattern
- Combining functions, modules, and previous knowledge effectively

Practical Applications

Real-world uses of functions and modules:

- **Data Processing:** Functions for cleaning, analyzing, and transforming data
- **Game Development:** Random number generation for unpredictability
- **Mathematical Computing:** Scientific calculations and simulations
- **Web Development:** Modular code organization and reusability
- **Automation:** Reusable tools for repetitive tasks

Connection to Next Lecture

Our next lecture will explore **advanced function features** and **file handling**. We'll build on today's function foundation to learn default parameters, keyword arguments, and reading/writing data files. The modular programming concepts you've learned today will be essential for managing larger programs and persistent data storage.

Additional Resources

Textbook References:

- Chapter 4.2-4.7: Function definition through math module
- Chapter 4.13: Scope rules and variable accessibility
- Figure 4.1: Function definition anatomy (page references in notebook)

Official Notebook Examples:

- `ch04/04_02.ipynb`: Basic function definition patterns
- `ch04/04_03.ipynb`: Multiple parameter functions (maximum example)
- `ch04/04_04.ipynb`: Random module usage and seeding
- `ch04/04_05.ipynb`: Game of chance case study
- `ch04/04_07.ipynb`: Math module functions and constants

Remember: Functions are the building blocks of larger programs. Master these concepts thoroughly, as they form the foundation for all advanced programming topics. Practice creating your own functions and experiment with different parameter combinations and return values.