Lecture 2 Handout

Introduction to Python Programming

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Deitel Chapter 2, Pages 49-72

Learning Objectives

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

1. **Create and manipulate** variables with meaningful names following Python conventions
2. **Work with** different data types (integers, floats, strings) and check their types
3. **Perform** arithmetic calculations using all Python operators with proper precedence
4. **Format output** professionally using f-strings and print() function features
5. **Get user input** and convert between data types as needed
6. **Make decisions** in programs using if statements and boolean expressions
7. **Compare values** using relational operators and understand truthiness
8. **Build complete programs** that solve real-world problems step by step

1 Today's Learning Journey

This lecture builds directly on the foundations from Lecture 1, taking you from basic Python concepts to writing interactive, decision-making programs. We'll follow a structured five-part journey:

Part I: Variables and Assignment (15 min)

- Creating and naming variables with proper conventions
- Understanding data types and memory concepts
- Using the type() function for verification
- Hands-on practice with personal data variables

Part II: Arithmetic Operations (15 min)

- Mastering all seven arithmetic operators
- Understanding operator precedence (PEMDAS)
- Using augmented assignment operators
- Building compound interest calculators

Part III: Input/Output and Strings (15 min)

- Advanced print() function features
- Professional f-string formatting techniques
- Getting and processing user input
- Creating interactive greeting programs

Part IV: Decision Making with if (15 min)

- Boolean values and comparison operators
- if statement syntax and Python indentation
- String comparisons and validation
- Building age verification and login systems

Part V: Objects and Wrap-up (15 min)

- Understanding Python's object model
- Dynamic typing and object references
- Preview of descriptive statistics applications
- Reviewing accomplishments and next steps

Today's Focus: Writing real Python programs that you can use immediately!

2 Review from Last Lecture

Before diving into new material, let's quickly review the key concepts from Lecture 1:

2.1 Programming Concepts

- **What is programming?** Problem solving with precise computer instructions
- **Why Python?** Readable, powerful, versatile, and extremely popular
- **Good programming language features:** Easy to learn, expressive syntax, large community

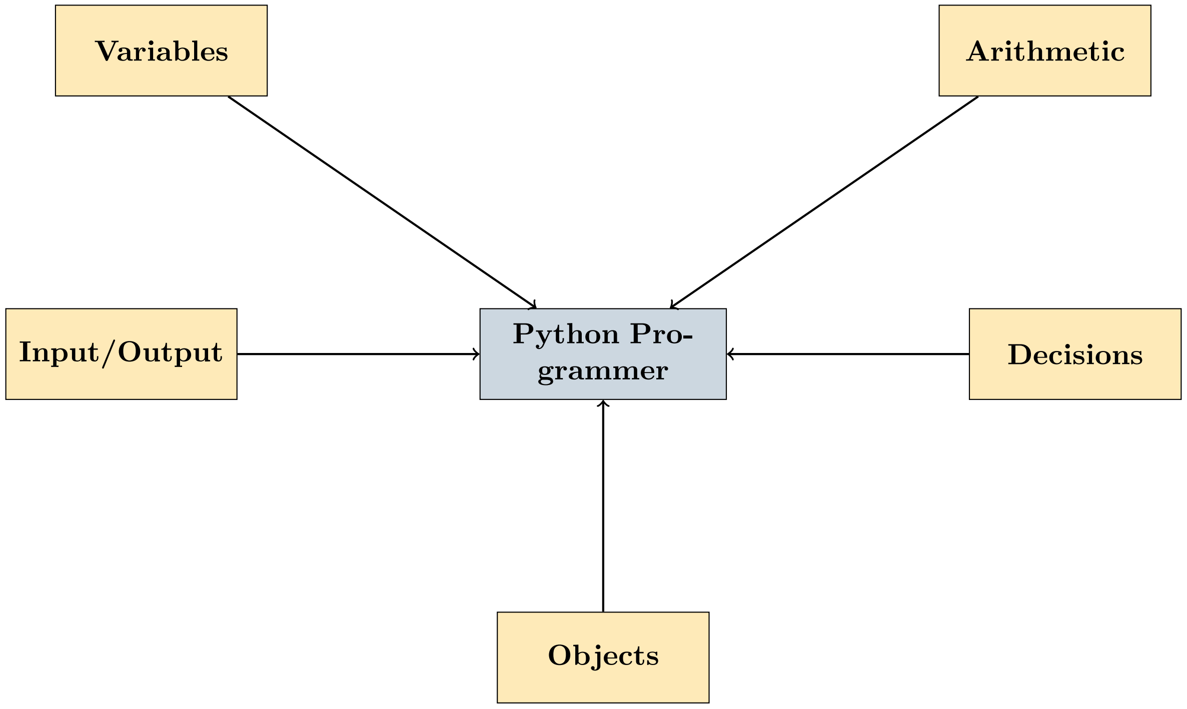
2.2 Hands-On Experience From Last Class

- Used Google Colab for cloud-based Python development
- Wrote ”Hello, World!” programs with proper syntax
- Created simple variables and performed basic calculations
- Explored Python as a powerful interactive calculator

This foundation gives us everything we need to build more sophisticated programs today!

3 Chapter 2 Overview - Your Python Foundation

Chapter 2 of the Deitel textbook focuses on the fundamental building blocks of Python programming. Think of today’s material as learning the essential tools that every Python programmer must master.



Today’s Mission: Transform from ”Python curious” to ”Python capable”

Each concept builds on the previous one, just like learning to drive a car - you need to master the fundamentals before you can navigate complex situations!

4 Part I: Variables and Assignment

4.1 Understanding Variables: Named Storage Containers

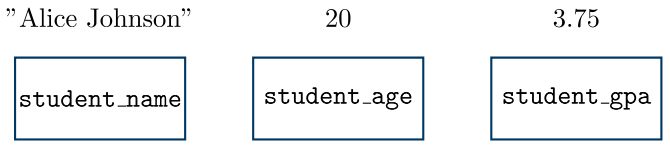
Variables are the foundation of all programming. Think of them as labeled boxes in a warehouse where you can store different types of information.

Key Concepts:

- Each variable has a **name** (identifier) that you choose
- Each variable stores a **value** of some type

- Values have a **type** (integer, string, float, boolean)
- Variables can be **changed** throughout your program (that's why they're called ”variable”)

Real-World Analogy: Variables are like labeled boxes in a warehouse. Each box has a label (the variable name) and contains something valuable (the data).



Computer Memory

Variables make programs flexible and reusable because you can change the values and your calculations will automatically update!

4.2 Creating Variables with Assignment Statements

The fundamental syntax for creating variables in Python is simple yet powerful:

```
variable_name = value
```

The equals sign (=) is the **assignment operator**. It means ”take the value on the right and store it in the variable named on the left.”

Important Note: The assignment operator (=) is NOT the same as mathematical equality. It's an instruction to store a value, not a statement that two things are equal.

Listing 1: Creating Variables with Different Data Types

```python
# Storing text (strings) - note the quotes are required
student_name = "Alice Johnson"
university = "Quinnipiac University"
major = "Computer Science"
course_code = "INF 605"

# Storing whole numbers (integers)
current_year = 2025
student_age = 20
credits_needed = 120
semester_number = 2

# Storing decimal numbers (floats)
student_gpa = 3.75
tuition_cost = 52890.00
coffee_price = 4.95
tax_rate = 0.08

# Storing True/False values (booleans)
is_enrolled = True
has_scholarship = False
is_on_campus = True
completed_prerequisites = False
```

Memory Concept: When you create a variable, Python allocates memory space and creates a reference from your variable name to that memory location.

4.3 Variable Naming Rules and Best Practices

Python has both **required rules** that must be followed (or your program won't work) and **recommended conventions** that make your code professional and readable.

4.3.1 Required Rules (Must Follow)

- Must start with a letter (a-z, A-Z) or underscore (_)
- Can contain letters, numbers (0-9), and underscores
- Cannot start with a number: `2names` is invalid
- Cannot contain spaces: `first name` is invalid
- Cannot use special characters: `student@email` is invalid
- Cannot use Python keywords: `if`, `for`, `while`, `class`, etc.

4.3.2 Recommended Style Guidelines (PEP 8)

Python has an official style guide called PEP 8 (Python Enhancement Proposal 8) that provides recommendations for writing clean, readable code:

- Use **snake_case**: all lowercase letters with underscores between words
- Use descriptive names: `total_price` not `tp`
- Avoid abbreviations: `temperature` not `temp`
- Use meaningful names: `user_age` not `x`
- Constants use ALL_CAPS: `MAX_ATTEMPTS = 3`

Examples of Good and Bad Variable Names:

- **Excellent:** `first_name`, `total_cost`, `is_valid_email`
- **Acceptable:** `name`, `cost`, `valid`
- **Poor:** `n`, `x`, `data`, `temp`
- **Invalid:** `2names`, `first-name`, `user@email`

4.4 Python is Case Sensitive!

This is a common source of errors for beginning programmers. Python treats these as completely different variables:

All Different Variables!

`student_name`	↔ `Student_Name`	↔ `STUDENT_NAME`
Variable 1	**Variable 2**	**Variable 3**

Common Beginner Mistake:

```python
# Create a variable
studentName = "Alice"

# Later try to use it (with different capitalization)
print(studentname)  # ERROR: NameError: name ’studentname’ is not defined
```

Best Practice: Choose one naming convention and stick to it consistently. Python recommends snake_case.

4.5 Python Data Types Explained

Python automatically determines the type of data based on how you write the value. This is called **dynamic typing** and makes Python very beginner-friendly.

4.5.1 1. Integers (int) - Whole Numbers

Integers represent whole numbers without decimal points:

```python
age = 20
year = 2025
temperature = -5
score = 0
population = 8000000
```

4.5.2 2. Floating-Point Numbers (float) - Decimal Numbers

Floats represent numbers with decimal points:

```python
gpa = 3.75
price = 29.99
temperature = 98.6
pi_approximation = 3.14159
percentage = 0.85
```

4.5.3 3. Strings (str) - Text Data

Strings represent text and must be enclosed in quotes (single or double):

```python
name = "Alice Johnson"
message = ’Hello, World!’  # Single quotes also work
address = "123 Main St, Hamden, CT"
empty_string = ""  # Valid empty string
course = "Introduction to Programming"
```

4.5.4 4. Booleans (bool) - True/False Values

Booleans represent logical true/false values:

```python
is_student = True       # Note: Capital T
has_license = False     # Note: Capital F
is_enrolled = True
completed_course = False
```

Key Point: Python automatically determines the type - you don’t need to declare it explicitly like in some other languages!

4.6 The type() Function - Checking Data Types

Python provides the `type()` function to check what type of data you're working with. This is incredibly useful for debugging and understanding your program's behavior.

Listing 2: Using the type() Function

```
# Create variables of different types
student_name = "Alice Johnson"
student_age = 20
student_gpa = 3.75
is_enrolled = True

# Check their types - Python shows the class name
print(type(student_name))    # <class 'str'>
print(type(student_age))     # <class 'int'>
print(type(student_gpa))     # <class 'float'>
print(type(is_enrolled))     # <class 'bool'>

# You can also check types of literal values
print(type(42))              # <class 'int'>
print(type(3.14))            # <class 'float'>
print(type("Hello"))         # <class 'str'>
print(type(True))            # <class 'bool'>
```

Why This Matters:

- Different types support different operations
- Helps debug when programs don't work as expected
- Some functions require specific types as input
- Understanding types prevents common errors

4.7 Hands-On Exercise: Personal Data Variables

Let's practice creating variables by building a personal information system:

Listing 3: Personal Data Variables Exercise - Complete Example

```
# Personal Information Variables
# Fill in your own information

# Text information (strings)
my_name = "Your Full Name Here"
my_major = "Your Major Here"
my_hometown = "Your Hometown, State"
favorite_color = "Your Favorite Color"

# Numerical information (integers and floats)
my_age = 20                    # Your age (integer)
current_year = 2025           # Current year
credits_completed = 30        # Credits completed so far
my_gpa = 3.5                  # Your GPA (float)
expected_graduation = 2027    # Expected graduation year

# Boolean (True/False) information
is_on_campus = True           # Do you live on campus?
has_car = False              # Do you have a car?
is_working = True            # Do you have a job?
```

```python
plays_sports = False          # Do you play sports?

# Display all information with professional formatting
print("=" * 50)
print("STUDENT INFORMATION SYSTEM")
print("=" * 50)
print()

# Basic information
print(f"Name: {my_name}")
print(f"Age: {my_age} years old")
print(f"Major: {my_major}")
print(f"Hometown: {my_hometown}")
print(f"Favorite Color: {favorite_color}")
print()

# Academic information
print("ACADEMIC INFORMATION:")
print(f"Current GPA: {my_gpa}")
print(f"Credits Completed: {credits_completed}")
print(f"Expected Graduation: {expected_graduation}")
print()

# Lifestyle information
print("LIFESTYLE INFORMATION:")
print(f"Lives on Campus: {is_on_campus}")
print(f"Has Car: {has_car}")
print(f"Currently Working: {is_working}")
print(f"Plays Sports: {plays_sports}")
print()

# Check data types (for learning)
print("DATA TYPE VERIFICATION:")
print(f"Name type: {type(my_name)}")
print(f"Age type: {type(my_age)}")
print(f"GPA type: {type(my_gpa)}")
print(f"On Campus type: {type(is_on_campus)}")

print("=" * 50)
print("System Complete!")
```

Extension Activities:

- Add more personal variables: birth month, number of siblings, etc.
- Calculate derived information: birth year, years until graduation
- Practice with different data types and see how they behave

5 Part II: Arithmetic Operations

5.1 Python's Complete Set of Arithmetic Operators

Python provides seven arithmetic operators that let you perform mathematical calculations. Understanding all of them and their proper usage is essential for any programming task.

Operator	Operation	Example	Result
+	Addition	`5 + 3`	8
-	Subtraction	`5 - 3`	2
*	Multiplication	`5 * 3`	15
**	Exponentiation	`5 ** 3`	125
/	Division (Float)	`5 / 3`	1.6667...
//	Floor Division	`5 // 3`	1
%	Modulus (Remainder)	`5 % 3`	2

Special Notes:

- `**` is exponentiation in Python, NOT ^ (which is used in some other languages)
- Regular division (`/`) always returns a float, even for whole number results
- Floor division (`//`) returns only the quotient (whole number part)
- Modulus (`%`) returns only the remainder

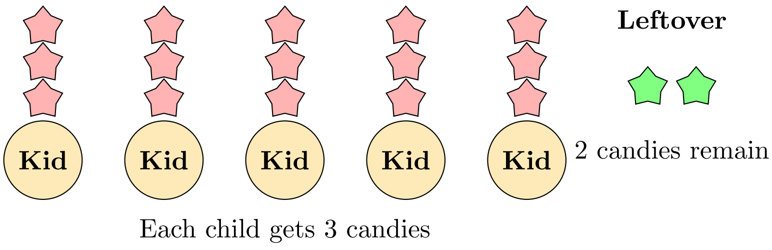
Listing 4: Testing All Arithmetic Operators

```python
# Comprehensive operator demonstration
a, b = 17, 5

print("ARITHMETIC OPERATORS DEMONSTRATION")
print(f"Working with a = {a} and b = {b}")
print("-" * 40)

print(f"Addition:       {a} + {b} = {a + b}")        # 22
print(f"Subtraction:    {a} - {b} = {a - b}")        # 12
print(f"Multiplication: {a} * {b} = {a * b}")        # 85
print(f"Exponentiation: {a} ** {b} = {a ** b}")      # 1419857 (17^5)
print(f"Division:       {a} / {b} = {a / b}")        # 3.4
print(f"Floor Division: {a} // {b} = {a // b}")      # 3 (quotient only)
print(f"Modulus:        {a} % {b} = {a % b}")        # 2 (remainder only)

print("\nVERIFICATION:")
print(f"Check floor division: {a // b} * {b} + {a % b} = {(a // b) * b + (a % b)}")
print(f"This equals our original number: {a}")
```

5.2 Understanding Floor Division and Modulus

Floor division and modulus are powerful operators that many beginners find confusing. Let's understand them with a practical example.

Real-World Example: Dividing 17 candies among 5 children



In Python:

- 17 // 5 = 3 (each child gets 3 candies - the quotient)
- 17 % 5 = 2 (2 candies are left over - the remainder)
- Verification: 3 × 5 + 2 = 17 ✓

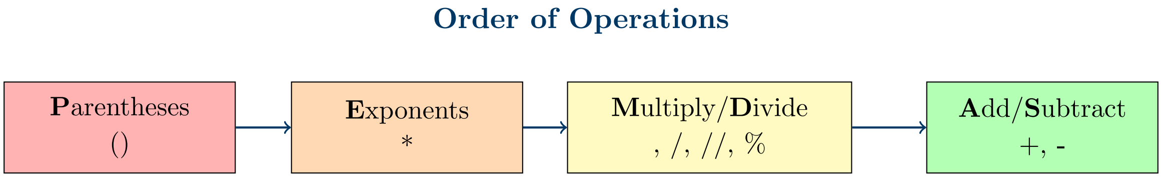
5.2.1 Practical Applications of Floor Division and Modulus

Listing 5: Practical Uses for Floor Division and Modulus

```python
# Time conversion example
total_minutes = 157

hours = total_minutes // 60       # How many complete hours?
remaining_minutes = total_minutes % 60  # How many minutes left?

print(f"{total_minutes} minutes equals {hours} hours and {
    remaining_minutes} minutes")
# Output: 157 minutes equals 2 hours and 37 minutes

# Check if a number is even or odd
number = 23
if number % 2 == 0:
    print(f"{number} is even")
else:
    print(f"{number} is odd")

# Cycle through a list (useful for arrays/lists later)
day_number = 23
days_of_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
     "Saturday", "Sunday"]
day_index = day_number % 7  # Will give us a number 0-6
print(f"Day {day_number} of the month falls on index {day_index}")

# Distribution problem
total_items = 100
containers = 7
items_per_container = total_items // containers
leftover_items = total_items % containers

print(f"Distribute {total_items} items into {containers} containers:")
print(f"Each container gets {items_per_container} items")
print(f"There are {leftover_items} items left over")
```

5.3 Operator Precedence - PEMDAS Rules

Python follows the standard mathematical order of operations, often remembered by the acronym PEMDAS:



Important Details:

- Operations of equal precedence are evaluated left to right
- Multiplication, division, floor division, and modulus have equal precedence
- Addition and subtraction have equal precedence (lower than multiplication/division)

Listing 6: Order of Operations Examples

```python
# Basic precedence examples
print("ORDER OF OPERATIONS EXAMPLES:")
print(f"2 + 3 * 4 = {2 + 3 * 4}")           # 14 (not 20!)
print(f"(2 + 3) * 4 = {(2 + 3) * 4}")       # 20

print(f"2 ** 3 * 4 = {2 ** 3 * 4}")         # 32 (exponent first)
print(f"2 * 3 ** 4 = {2 * 3 ** 4}")         # 162 (exponent first)

# Left-to-right for same precedence
print(f"20 / 4 * 5 = {20 / 4 * 5}")         # 25.0 (not 1.0!)
print(f"20 / (4 * 5) = {20 / (4 * 5)}")     # 1.0

# Complex expressions
print(f"10 - 3 * 2 + 5 = {10 - 3 * 2 + 5}") # 11
print(f"(10 - 3) * (2 + 5) = {(10 - 3) * (2 + 5)}") # 49
```

Best Practice: Use parentheses to make your intentions clear, even when not required:

- Unclear: `price * quantity + tax`
- Clear: `(price * quantity) + tax`

5.4 Augmented Assignment Operators

Augmented assignment operators provide a shortcut for modifying variables. They combine an arithmetic operation with assignment in a single step.

Operator	Equivalent Long Form	Meaning
`+=`	`x = x + 5`	Add and assign
`-=`	`x = x - 5`	Subtract and assign
`*=`	`x = x * 5`	Multiply and assign
`/=`	`x = x / 5`	Divide and assign
`**=`	`x = x ** 5`	Exponentiate and assign
`//=`	`x = x // 5`	Floor divide and assign
`%=`	`x = x % 5`	Modulus and assign

Listing 7: Augmented Assignment Examples - Banking System

```python
# Bank account balance simulation
print("BANKING SYSTEM SIMULATION")
print("=" * 30)

account_balance = 1000.00
print(f"Starting balance: ${account_balance:.2f}")

# Deposit money (addition)
deposit_amount = 250.00
account_balance += deposit_amount
print(f"After depositing ${deposit_amount}: ${account_balance:.2f}")

```

```python
# Pay monthly fee (subtraction)
monthly_fee = 15.00
account_balance -= monthly_fee
print(f"After monthly fee of ${monthly_fee}: ${account_balance:.2f}")

# Interest compounds monthly (multiplication)
monthly_interest_rate = 1.015  # 1.5% monthly interest
account_balance *= monthly_interest_rate
print(f"After 1.5% interest: ${account_balance:.2f}")

# Withdraw half for emergency (division)
account_balance /= 2
print(f"After withdrawing half: ${account_balance:.2f}")

print("=" * 30)
print("Banking simulation complete!")
```

Benefits of Augmented Assignment:

- More concise and readable code
- Less typing and fewer opportunities for typos
- Shows clear intent to modify a variable
- Commonly used in professional code

5.5 Hands-On Exercise: Compound Interest Calculator

Let's build a practical financial calculator that demonstrates all the arithmetic concepts we've learned:

Listing 8: Complete Compound Interest Calculator

```python
# Compound Interest Calculator
# Formula: Final Amount = Principal $\times$ (1 + rate)^years

print("=" * 60)
print("COMPOUND INTEREST INVESTMENT CALCULATOR")
print("=" * 60)

# Investment parameters
principal = 1000.00         # Initial investment amount
annual_rate = 0.05        # 5% annual interest rate
years = 10                # Investment period in years

# Display investment details
print("INVESTMENT PARAMETERS:")
print(f"Principal Amount: ${principal:.2f}")
print(f"Annual Interest Rate: {annual_rate * 100}%")
print(f"Investment Period: {years} years")
print()

# Method 1: Using the compound interest formula directly
print("METHOD 1: Direct Formula Calculation")
final_amount = principal * (1 + annual_rate) ** years
interest_earned = final_amount - principal
total_growth_percentage = (final_amount / principal - 1) * 100

```

```python
print(f"Final Amount: ${final_amount:.2f}")
print(f"Interest Earned: ${interest_earned:.2f}")
print(f"Total Growth: {total_growth_percentage:.1f}%")
print()

# Method 2: Year-by-year simulation to show compound growth
print("METHOD 2: Year-by-Year Growth Simulation")
print("Year | Balance")
print("-" * 20)

current_amount = principal
print(f"  0  | ${current_amount:.2f}")

for year in range(1, years + 1):
    yearly_interest = current_amount * annual_rate
    current_amount += yearly_interest
    print(f" {year:2d}  | ${current_amount:.2f}")

print("-" * 20)
print(f"Final amount matches: ${current_amount:.2f}")
print()

# Comparison with different interest rates
print("RATE COMPARISON ANALYSIS:")
print("Rate | 10-Year Final Amount | Total Return")
print("-" * 45)

for rate in [0.03, 0.05, 0.07, 0.10]:
    final = principal * (1 + rate) ** years
    return_amount = final - principal
    return_percentage = (final / principal - 1) * 100

    print(f"{rate*100:4.1f}% | ${final:15,.2f} | ${return_amount:7,.2f} ({return_percentage:.1f}%)")

print("=" * 60)
print("Investment analysis complete!")
```

5.6 Math Module Preview

While Python's built-in arithmetic operators handle most calculations, the math module provides advanced mathematical functions for more complex operations:

Listing 9: Math Module Functions

```python
import math

# Mathematical constants
print(f"Pi: {math.pi}")                    # 3.14159...
print(f"Euler's number (e): {math.e}")     # 2.71828...

# Common functions
print(f"Square root of 16: {math.sqrt(16)}")        # 4.0
print(f"2 to the power of 8: {math.pow(2, 8)}")     # 256.0
print(f"Ceiling of 4.3: {math.ceil(4.3)}")          # 5
print(f"Floor of 4.7: {math.floor(4.7)}")           # 4

# Trigonometric functions (angles in radians)
```

```
print(f"sin($\pi$/2): {math.sin(math.pi/2)}")          # 1.0
print(f"cos(0): {math.cos(0)}")                       # 1.0

# Convert between degrees and radians
angle_degrees = 45
angle_radians = math.radians(angle_degrees)
print(f"{angle_degrees} degrees = {angle_radians:.4f} radians")
```

6 Part III: Input/Output and Strings

6.1 Advanced print() Function Features

The print() function is more powerful than it first appears. Understanding its full capabilities lets you create professional-looking output.

Listing 10: Advanced print() Function Techniques

```
# Basic printing
print("Hello, World!")

# Printing multiple items (automatic space separation)
name = "Alice"
age = 20
gpa = 3.75
print("Student:", name, "Age:", age, "GPA:", gpa)
# Output: Student: Alice Age: 20 GPA: 3.75

# Custom separator between items
print("Apple", "Banana", "Cherry", sep=", ")
# Output: Apple, Banana, Cherry

print("2025", "01", "27", sep="-")
# Output: 2025-01-27

# Custom ending (default is newline \n)
print("Loading", end="...")
print("Complete!")
# Output: Loading...Complete!

# Print to different destinations
import sys
print("Normal message")                      # Goes to standard output
print("Error message", file=sys.stderr)     # Goes to error stream

# No separator between items
print("A", "B", "C", sep="")                # Output: ABC

# Multiple lines with escape characters
print("Line 1\nLine 2\nLine 3")            # \n creates new lines
print("Name:\tAlice\nAge:\t20")             # \t creates tab spacing
```

Key print() Parameters:

- `sep=` controls what goes between multiple items
- `end=` controls what goes at the end of the print statement
- `file=` controls where the output goes

6.2 Understanding Strings - Text Data in Detail

Strings are sequences of characters that represent text data. Understanding how to work with strings is crucial for interactive programming.

6.2.1 Different Ways to Create Strings

Listing 11: String Creation Methods

```
# Single quotes
name1 = 'Alice Johnson'
message1 = 'Hello, World!'

# Double quotes (Python's preference)
name2 = "Bob Smith"
message2 = "Welcome to Python!"

# When to use each type
apostrophe_string = "It's a beautiful day"          # Use double quotes for apostrophes
quote_string = 'He said "Hello" to everyone'        # Use single quotes for internal quotes

# Triple quotes for multi-line strings
long_message = """This is a long message
that spans multiple lines.
It preserves line breaks and spacing."""

# Empty strings are valid
empty_string = ""
empty_string2 = ''
```

6.2.2 Escape Characters for Special Formatting

Listing 12: Escape Characters in Strings

```
# Common escape sequences
print("Line 1\nLine 2")            # \n creates new line
print("Name:\tAge:\tGPA:")         # \t creates tab spacing
print("This is a \"quoted\" word") # \" includes quote in string
print('It\'s a nice day')          # \' includes apostrophe in single-quoted string
print("Path: C:\\Users\\Alice")    # \\ creates literal backslash

# Raw strings (prefix with r) treat backslashes literally
file_path = r"C:\Users\Alice\Documents\file.txt"
print(file_path)

# Unicode characters
print("Python is fun! \u2764")   # \u2764 is heart symbol (heart emoji)
print("Greek letter pi: \u03C0") # \u03C0 is $\pi$
```

6.3 F-String Formatting - The Modern Way

F-strings (formatted string literals) are Python's most modern and powerful way to include variables in strings. They're fast, readable, and incredibly flexible.

6.3.1 Basic F-String Syntax and Usage

Listing 13: Basic F-String Examples

```python
# Basic f-string syntax: f"text {variable} more text"
name = "Alice Johnson"
age = 20
university = "Quinnipiac University"

# Simple variable insertion
print(f"Hello, my name is {name}")
print(f"I am {age} years old")
print(f"I attend {university}")

# Multiple variables in one string
print(f"My name is {name}, I’m {age} years old, and I attend {university}.")

# Expressions inside braces
width = 10
height = 5
print(f"Rectangle area: {width * height} square units")
print(f"Rectangle perimeter: {2 * (width + height)} units")

# Function calls inside f-strings
import math
radius = 7
print(f"Circle area: {math.pi * radius**2:.2f}")
```

6.3.2 Number Formatting with F-Strings

Listing 14: Advanced F-String Number Formatting

```python
# Decimal places formatting
price = 29.99567
print(f"Price: ${price:.2f}")              # $29.96 (2 decimal places)
print(f"Price: ${price:.4f}")              # $29.9957 (4 decimal places)

# Percentage formatting
success_rate = 0.847
print(f"Success rate: {success_rate:.1%}")  # Success rate: 84.7%
print(f"Success rate: {success_rate:.2%}")  # Success rate: 84.70%

# Integer formatting with thousands separator
population = 1234567
salary = 75000
print(f"Population: {population:,}")        # Population: 1,234,567
print(f"Annual salary: ${salary:,}")        # Annual salary: $75,000

# Scientific notation
large_number = 1234567890
print(f"Scientific: {large_number:.2e}")    # Scientific: 1.23e+09

# Field width and alignment
name1, name2 = "Alice", "Bob"
score1, score2 = 95, 87

```

```python
# Right-aligned in specified width
print(f"{name1:>10}: {score1:>3}")         # Right-aligned
print(f"{name2:>10}: {score2:>3}")

# Left-aligned
print(f"{name1:<10}: {score1}")             # Left-aligned
print(f"{name2:<10}: {score2}")

# Center-aligned
print(f"{'SCORES':^20}")                   # Centered in 20 characters
```

6.3.3 Professional Output Formatting

Listing 15: Professional Receipt Example with F-Strings

```python
# Professional receipt formatting example
product = "Wireless Headphones"
price = 89.99
quantity = 2
tax_rate = 0.0825

# Calculate totals
subtotal = price * quantity
tax_amount = subtotal * tax_rate
total = subtotal + tax_amount

# Create professional receipt
print("=" * 40)
print(f"{'ELECTRONICS STORE RECEIPT':^40}")
print("=" * 40)
print(f"Date: {'2025-01-27':>32}")
print()

print(f"{'Item':<20} {'Qty':>5} {'Price':>8} {'Total':>8}")
print("-" * 40)
print(f"{product:<20} {quantity:>5} ${price:>7.2f} ${subtotal:>7.2f}")
print()

print(f"{'Subtotal:':<32} ${subtotal:>7.2f}")
print(f"{'Tax ({:.1%}):':<32} ${tax_amount:>7.2f}".format(tax_rate))
print("-" * 40)
print(f"{'TOTAL:':<32} ${total:>7.2f}")
print("=" * 40)
print("Thank you for your business!")
```

6.4 Getting User Input with input() Function

The input() function transforms your programs from static calculators into interactive applications that respond to user needs.

6.4.1 Basic Input Function Usage

Listing 16: Basic Input Function Examples

```python
# Basic input - always returns a string
name = input("What is your name? ")
```

```python
print(f"Hello, {name}!")

# Input with descriptive prompts
favorite_color = input("What's your favorite color? ")
print(f"That's cool! {favorite_color} is a beautiful color.")

# Multiple inputs
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
print(f"Nice to meet you, {first_name} {last_name}!")
```

6.4.2 Type Conversion for Numeric Input

Since input() always returns a string, you must convert to numbers for mathematical operations:

Listing 17: Converting Input to Numbers

```python
# Getting numbers requires type conversion
print("GRADE CALCULATOR")
print("-" * 20)

# Method 1: Convert after input
age_string = input("How old are you? ")
age = int(age_string)
print(f"Wow, {age} is a great age!")

# Method 2: Convert during input (more common)
height = float(input("Enter your height in meters: "))
weight = float(input("Enter your weight in kg: "))

# Calculate BMI
bmi = weight / (height ** 2)
print(f"Your BMI is {bmi:.1f}")

# Multiple numeric inputs example
print("\nGRADE POINT AVERAGE CALCULATOR")
print("Enter your grades for 4 classes:")

grade1 = float(input("Class 1 grade: "))
grade2 = float(input("Class 2 grade: "))
grade3 = float(input("Class 3 grade: "))
grade4 = float(input("Class 4 grade: "))

# Calculate average
average = (grade1 + grade2 + grade3 + grade4) / 4
print(f"Your GPA is: {average:.2f}")
```

6.5 Type Conversion Functions Explained

Understanding type conversion is crucial for working with user input and different data types.

Function	Purpose	Example
`int()`	Convert to integer	`int("42")` → 42
`float()`	Convert to decimal	`float("3.14")` → 3.14
`str()`	Convert to string	`str(42)` → ”42”
`bool()`	Convert to boolean	`bool(1)` → True

6.5.1 Common Conversion Scenarios and Pitfalls

Listing 18: Type Conversion Examples and Error Handling

```python
# Safe conversions
print("SAFE TYPE CONVERSIONS:")
number_string = "42"
decimal_string = "3.14159"
integer_number = 100

print(f"String to int: int('{number_string}') = {int(number_string)}")
print(f"String to float: float('{decimal_string}') = {float(decimal_string)}")
print(f"Int to string: str({integer_number}) = '{str(integer_number)}'")

# Boolean conversions
print("\nBOOLEAN CONVERSION RULES:")
print(f"bool(0) = {bool(0)}")              # False
print(f"bool(42) = {bool(42)}")            # True
print(f"bool('') = {bool('')}")            # False (empty string)
print(f"bool('hi') = {bool('hi')}")        # True (non-empty string)
print(f"bool([]) = {bool([])}")            # False (empty list)

# Common conversion errors (commented out to avoid crashes)
# print(int("hello"))         # ValueError: invalid literal
# print(float("3.14.15"))     # ValueError: could not convert
# print(int("3.14"))          # ValueError: invalid literal (use float first)

# Safe conversion with error checking (preview of try/except)
user_input = "not_a_number"
try:
    number = int(user_input)
    print(f"Successfully converted: {number}")
except ValueError:
    print(f"Cannot convert '{user_input}' to integer")
```

6.6 Hands-On Exercise: Interactive Greeting Program

Let's create a comprehensive interactive program that demonstrates all input/output concepts:

Listing 19: Complete Interactive Greeting Program

```python
# Interactive Personal Greeting Generator
print("=" * 60)
print("WELCOME TO THE PERSONAL GREETING GENERATOR")
print("=" * 60)
print("Let's get to know you better!")
print()

# Collect personal information
print("PERSONAL INFORMATION:")
first_name = input("What's your first name? ")
last_name = input("What's your last name? ")
age = int(input("How old are you? "))
hometown = input("Where are you from? ")
favorite_hobby = input("What's your favorite hobby? ")
```

```python

# Collect academic information
print("\nACADEMIC INFORMATION:")
university = input("What university do you attend? ")
major = input("What's your major? ")
year_in_school = input("What year are you (freshman, sophomore, etc.)? ")

# Calculate derived information
current_year = 2025
birth_year = current_year - age
decade = birth_year // 10 * 10  # Round down to nearest decade
days_alive_approx = age * 365   # Approximate days alive

# Create personalized greeting with professional formatting
print("\n" + "=" * 60)
print(f"{'PERSONAL PROFILE GENERATED':^60}")
print("=" * 60)
print()

print(f"Hello, {first_name} {last_name}!")
print(f"It's great to meet a {age}-year-old {year_in_school} from {hometown}.")
print()

print("BACKGROUND ANALYSIS:")
print(f"$\bullet$ Born approximately in {birth_year}")
print(f"$\bullet$ You're part of the {decade}s generation")
print(f"$\bullet$ You've been alive for roughly {days_alive_approx:,} days")
print()

print("ACADEMIC PROFILE:")
print(f"$\bullet$ University: {university}")
print(f"$\bullet$ Major: {major}")
print(f"$\bullet$ Academic Level: {year_in_school.title()}")
print()

print("PERSONAL INTERESTS:")
print(f"$\bullet$ Favorite Hobby: {favorite_hobby}")
print(f"$\bullet$ {favorite_hobby.title()} is an excellent way to spend free time!")
print()

# Generate personalized messages based on age
print("PERSONALIZED INSIGHTS:")
if age < 18:
    print("$\bullet$ You're still in high school - exciting times ahead!")
elif age < 22:
    print("$\bullet$ You're in the prime college years - make the most of it!")
elif age < 25:
    print("$\bullet$ You're entering the professional world - great opportunities await!")
else:
    print("$\bullet$ You bring valuable life experience to your studies!")

print()
print("LOCATION INSIGHT:")
if "New York" in hometown or "NY" in hometown:
    print("$\bullet$ The Big Apple! You know how to handle fast-paced environments.")
elif "California" in hometown or "CA" in hometown:
    print("$\bullet$ West Coast vibes! You probably appreciate innovation and tech.")
elif "Texas" in hometown or "TX" in hometown:
    print("$\bullet$ Everything’s bigger in Texas, including opportunities!")
else:
    print(f"$\bullet$ {hometown} sounds like a wonderful place to grow up!")

print()
print("=" * 60)
print(f"Thanks for sharing, {first_name}! Welcome to our community!")
print("=" * 60)
```

6.7 Building an Interactive Calculator

Let’s combine all our input/output knowledge to create a fully functional calculator:

Listing 20: Interactive Calculator Program

```python
# Interactive Python Calculator
print("=" * 50)
print("PYTHON INTERACTIVE CALCULATOR")
print("=" * 50)
print("Available operations: +, -, *, /, **, //, %")
print()

# Get user input
first_number = float(input("Enter the first number: "))
operation = input("Enter operation (+, -, *, /, **, //, %): ")
second_number = float(input("Enter the second number: "))

print()
print(f"Calculating: {first_number} {operation} {second_number}")
print("-" * 30)

# Perform calculation based on operation
if operation == "+":
    result = first_number + second_number
    operation_name = "Addition"
elif operation == "-":
    result = first_number - second_number
    operation_name = "Subtraction"
elif operation == "*":
    result = first_number * second_number
    operation_name = "Multiplication"
elif operation == "/":
    if second_number != 0:
        result = first_number / second_number
        operation_name = "Division"
    else:
        result = "Error: Division by zero!"
        operation_name = "Division"
elif operation == "**":
    result = first_number ** second_number
    operation_name = "Exponentiation"
elif operation == "//":
    if second_number != 0:
        result = first_number // second_number
        operation_name = "Floor Division"
    else:
        result = "Error: Division by zero!"
        operation_name = "Floor Division"
elif operation == "%":
    if second_number != 0:
        result = first_number % second_number
        operation_name = "Modulus (Remainder)"
    else:
        result = "Error: Division by zero!"
        operation_name = "Modulus"
else:
    result = "Error: Invalid operation!"
    operation_name = "Unknown Operation"

# Display result
print(f"Operation: {operation_name}")
if isinstance(result, (int, float)):
    print(f"Result: {result}")
    if isinstance(result, float) and result.is_integer():
        print(f"Result (as integer): {int(result)}")
else:
    print(f"Result: {result}")

print("=" * 50)
print("Calculator session complete!")
```

7 Part IV: Decision Making with if

7.1 Boolean Values - The Foundation of Decisions

Boolean values are the foundation of all decision-making in programming. Understanding how they work is crucial for writing interactive and intelligent programs.

7.1.1 Boolean Data Type Explained

The boolean data type has only two possible values:

- `True` - Represents a positive, yes, or valid condition
- `False` - Represents a negative, no, or invalid condition

Important Notes:

- Boolean values are always capitalized: `True` and `False`
- They are used for yes/no, on/off, valid/invalid decisions
- Every condition in an if statement must evaluate to True or False

Listing 21: Boolean Variables and Expressions

```python
# Boolean literals
is_student = True
has_scholarship = False
is_on_campus = True
completed_prerequisites = False

# Boolean expressions (evaluate to True or False)
age = 20
is_adult = age >= 18              # True
can_vote = age >= 18              # True
can_drink = age >= 21             # False

# String expressions that return boolean
name = "Alice"
has_long_name = len(name) > 10    # False
starts_with_a = name.startswith("A")  # True

print("BOOLEAN VALUES DEMONSTRATION:")
print(f"is_student: {is_student}")
print(f"is_adult: {is_adult}")
print(f"can_vote: {can_vote}")
print(f"can_drink: {can_drink}")
print(f"has_long_name: {has_long_name}")
print(f"starts_with_a: {starts_with_a}")
```

7.2 Comparison Operators for Decision Making

Comparison operators let you compare values and return boolean results. They are essential for creating conditions in if statements.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
<	Less than	3 < 5	True
<=	Less than or equal	5 <= 5	True
>	Greater than	7 > 5	True
>=	Greater than or equal	5 >= 3	True

7.2.1 Working with Variables in Comparisons

Listing 22: Comparison Operators with Variables

```python
# Numeric comparisons
age = 20
minimum_voting_age = 18
drinking_age = 21

print("NUMERIC COMPARISONS:")
print(f"age = {age}")
print(f"age >= minimum_voting_age: {age >= minimum_voting_age}")  #
   True
print(f"age < drinking_age: {age < drinking_age}")               # True
print(f"age == 25: {age == 25}")                                 #
   False
print(f"age != 25: {age != 25}")                                 # True
```

```python

# String comparisons
name1 = "Alice"
name2 = "Bob"
name3 = "Alice"

print("\nSTRING COMPARISONS:")
print(f"name1 == name3: {name1 == name3}")       # True
print(f"name1 != name2: {name1 != name2}")       # True
print(f"name1 < name2: {name1 < name2}")         # True (alphabetical)

# Grade comparison example
test_score = 85
passing_score = 60
excellent_score = 90

print("\nGRADE ANALYSIS:")
print(f"Test score: {test_score}")
print(f"Passed: {test_score >= passing_score}")
print(f"Excellent: {test_score >= excellent_score}")
print(f"Failed: {test_score < passing_score}")
```

Common Mistake: Using assignment (=) instead of equality (==)

- Wrong: `if age = 18:` (assignment)
- Correct: `if age == 18:` (comparison)

7.3 The if Statement - Basic Syntax and Structure

The if statement is the foundation of program logic. It allows your program to make decisions and execute different code based on conditions.

7.3.1 Basic if Statement Structure

```
if condition:
statement_to_execute
another_statement
```

Key Points:

- The condition must evaluate to True or False
- Colon (:) is required after the condition
- Indented code block executes only if condition is True
- Python uses 4 spaces for indentation (standard)

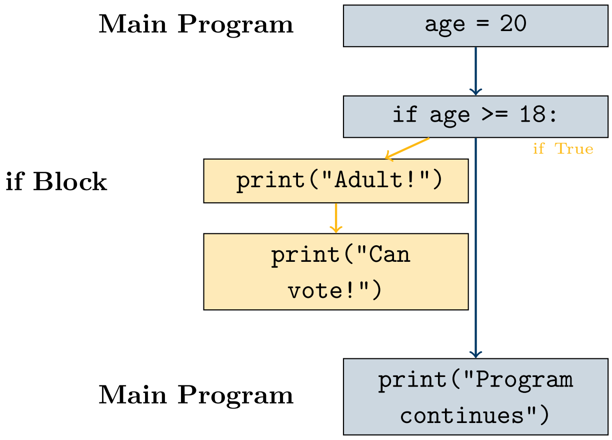
Listing 23: Basic if Statement Examples

```python
# Simple if statement
age = 20
if age >= 18:
    print("You are an adult!")
    print("You can vote in elections!")

# if statement with calculations
```

```python
score = 95
if score >= 90:
    print("Excellent work!")
    print("You earned an A grade!")
    letter_grade = "A"
    gpa_points = 4.0

# Multiple separate if statements
temperature = 75
if temperature > 80:
    print("It's hot outside!")
    print("Consider wearing light clothes.")

if temperature < 60:
    print("It's cool outside!")
    print("You might want a jacket.")

if temperature >= 60 and temperature <= 80:
    print("Perfect weather!")
    print("Great day to be outdoors.")
```

7.4 Understanding Python Indentation

Indentation is not just for readability in Python - it's how Python determines which statements belong together in code blocks.



7.4.1 Indentation Rules and Common Errors

Python Indentation Rules:

- Use 4 spaces for each indentation level (Python standard)
- All statements at the same level must have identical indentation
- if statements, loops, and functions all require indentation
- Never mix tabs and spaces (use spaces consistently)

Common Indentation Errors:

Listing 24: Indentation Errors to Avoid

```python
# ERROR: Missing indentation
age = 20
```

```python
if age >= 18:
print("You are an adult!")  # IndentationError: expected an indented block

# ERROR: Inconsistent indentation
if age >= 18:
    print("You are an adult!")     # 4 spaces
        print("You can vote!")     # 8 spaces - IndentationError

# CORRECT: Consistent indentation
if age >= 18:
    print("You are an adult!")     # 4 spaces
    print("You can vote!")         # 4 spaces
    print("Welcome to adulthood!") # 4 spaces
```

7.4.2 Students Often Struggle Here: Advanced Indentation Tips

Why Students Find Indentation Difficult:

- **Coming from other languages:** Many languages use {} instead of indentation
- **Invisible characters:** Spaces and tabs look the same but aren't!
- **Inconsistency:** Mixing different amounts of spaces
- **Nested structures:** Multiple levels of indentation can be confusing

Pro Tips for Mastering Indentation:

1. **Set up your editor:** Configure to show spaces/tabs visually
2. **Use spaces only:** Never mix tabs and spaces
3. **Be consistent:** Always use exactly 4 spaces per level
4. **Check alignment:** All statements at same level must align perfectly
5. **Practice nested structures:** Understand how multiple levels work

Listing 25: Advanced Indentation: Nested if Statements

```python
# Multiple indentation levels example
age = 22
has_license = True
temperature = 85

if age >= 18:
    print("You are an adult!")                   # Level 1: 4 spaces
    if has_license:
        print("You can drive!")                  # Level 2: 8 spaces
        if temperature > 80:
            print("Drive with AC on!")           # Level 3: 12 spaces
            print("Stay cool!")                  # Level 3: 12 spaces
        print("Drive safely!")                   # Level 2: 8 spaces
    print("Enjoy your adult privileges!")        # Level 1: 4 spaces
print("This always runs - no indentation")      # Main level: 0 spaces
```

Visual Indentation Guide:

Level	Spaces	Example
Main program	0	`print("Hello")`
if block	4	`    print("Inside if")`
Nested if	8	`        print("Nested")`
Triple nested	12	`            print("Deep")`

Red Flags - Fix These Immediately:

- Lines that should be indented but aren't
- Inconsistent spacing within the same block
- Mixing tabs and spaces (invisible but causes errors!)
- Copy-pasting code without checking indentation

7.5 String Comparisons and Validation

String comparisons are essential for user input validation and creating interactive programs.

7.5.1 Case-Sensitive String Comparisons

Listing 26: String Comparison Examples

```python
# Case-sensitive string comparison
name1 = "Alice"
name2 = "alice"
name3 = "ALICE"

print("CASE SENSITIVITY DEMONSTRATION:")
print(f"name1 == name2: {name1 == name2}")  # False
print(f"name1 == name3: {name1 == name3}")  # False
print(f"name2 == name3: {name2 == name3}")  # False

# Making case-insensitive comparisons
user_input = "YES"
if user_input.lower() == "yes":
    print("User confirmed!")

# Multiple acceptable responses
response = input("Do you want to continue? (yes/y/Y): ")
if response.lower() in ["yes", "y"]:
    print("Continuing program...")
else:
    print("Program stopped.")

# Email validation example
email = input("Enter your email: ")
if "@" in email and "." in email:
    print("Email format looks valid")
else:
    print("Please enter a valid email address")
```

7.5.2 String Methods for Validation

Listing 27: String Validation Methods

```
# Password strength checking
password = input("Create a password: ")

print("PASSWORD ANALYSIS:")
print(f"Length >= 8 characters: {len(password) >= 8}")
print(f"Contains only letters/numbers: {password.isalnum()}")
print(f"All uppercase: {password.isupper()}")
print(f"All lowercase: {password.islower()}")
print(f"Contains digits: {any(c.isdigit() for c in password)}")

# Name validation
name = input("Enter your name: ")
if name.isalpha():
    print("Name contains only letters - good!")
else:
    print("Name should contain only letters")

# Phone number validation (simple)
phone = input("Enter phone number (digits only): ")
if phone.isdigit() and len(phone) == 10:
    print("Phone number format is valid")
else:
    print("Please enter exactly 10 digits")
```

7.6 Hands-On Exercise: Age Verification System

Let's build a comprehensive age verification program that demonstrates all the concepts we've learned:

Listing 28: Complete Age Verification System

```
# Comprehensive Age Verification System
print("=" * 60)
print("ADVANCED AGE VERIFICATION SYSTEM")
print("=" * 60)
print("Let's see what you're eligible for based on your age!")
print()

# Get user information with validation
while True:
    try:
        name = input("What's your name? ")
        if name.strip():  # Check if name is not empty
            break
        else:
            print("Please enter a valid name.")
    except:
        print("Please enter a valid name.")

while True:
    try:
        age = int(input("How old are you? "))
        if age >= 0 and age <= 120:  # Reasonable age range
            break
        else:
            print("Please enter a valid age between 0 and 120.")
    except ValueError:
```

```python
        print("Please enter a number for your age.")

print(f"\nHello, {name}! Let's see what you can do at age {age}:")
print("=" * 60)

# Age-based eligibility checks
print("ELIGIBILITY ANALYSIS:")

# Basic age categories
if age < 13:
    print("[Child emoji] You're a kid - enjoy your childhood!")
    category = "Child"
elif age < 20:
    print("[Teen emoji] You're a teenager - exciting times ahead!")
    category = "Teenager"
elif age < 65:
    print("[Adult emoji] You're an adult - lots of opportunities!")
    category = "Adult"
else:
    print("[Senior emoji] You're a senior - wisdom and experience!")
    category = "Senior"

print(f"Age Category: {category}")
print()

# Specific privileges and responsibilities
print("WHAT YOU CAN DO:")
eligibilities = []

if age >= 5:
    eligibilities.append("$\checkmark$ Attend elementary school")
if age >= 13:
    eligibilities.append("$\checkmark$ Have social media accounts (with parent permission)")
if age >= 14:
    eligibilities.append("$\checkmark$ Work part-time jobs (with restrictions)")
if age >= 16:
    eligibilities.append("$\checkmark$ Get a driver's license")
    eligibilities.append("$\checkmark$ Work more flexible hours")
if age >= 17:
    eligibilities.append("$\checkmark$ Join the military (with parent consent)")
if age >= 18:
    eligibilities.append("$\checkmark$ Vote in elections")
    eligibilities.append("$\checkmark$ Sign legal contracts")
    eligibilities.append("$\checkmark$ Get married without consent")
    eligibilities.append("$\checkmark$ Join the military")
if age >= 21:
    eligibilities.append("$\checkmark$ Purchase and consume alcohol")
    eligibilities.append("$\checkmark$ Rent a car more easily")
if age >= 25:
    eligibilities.append("$\checkmark$ Run for U.S. House of Representatives")
    eligibilities.append("$\checkmark$ Get better car insurance rates")
if age >= 30:
    eligibilities.append("$\checkmark$ Run for U.S. Senate")
```

```python
if age >= 35:
    eligibilities.append("$\checkmark$  Run for President of the United States")
if age >= 65:
    eligibilities.append("$\checkmark$  Eligible for Medicare")
    eligibilities.append("$\checkmark$  Eligible for full Social Security benefits")

# Display eligibilities
for eligibility in eligibilities:
    print(eligibility)

if not eligibilities:
    print("$\bullet$ Keep growing! More privileges await as you get older!")

print()

# Special messages based on age milestones
print("SPECIAL MESSAGES:")
if age == 16:
    print("[Party emoji] Happy 16th! Time to think about driving!")
elif age == 18:
    print("[Party emoji] Happy 18th! You’re officially an adult!")
elif age == 21:
    print("[Party emoji] Happy 21st! Enjoy responsibly!")
elif age == 65:
    print("[Party emoji] Happy 65th! Retirement planning time!")
elif age >= 100:
    print("[Party emoji] Wow! You’re a centenarian! Congratulations!")

# Calculate some fun facts
current_year = 2025
birth_year = current_year - age
days_alive = age * 365  # Approximate
next_milestone = None

if age < 16:
    next_milestone = 16
elif age < 18:
    next_milestone = 18
elif age < 21:
    next_milestone = 21
elif age < 25:
    next_milestone = 25

print()
print("FUN FACTS ABOUT YOUR AGE:")
print(f"$\bullet$ You were born approximately in {birth_year}")
print(f"$\bullet$ You’ve been alive for roughly {days_alive:,} days")
if next_milestone:
    years_to_milestone = next_milestone - age
    print(f"$\bullet$ Only {years_to_milestone} year(s) until you turn {next_milestone}!")

print("=" * 60)
print(f"Thanks for using our system, {name}!")
print("Remember: Age is just a number, but it determines a lot!")
print("=" * 60)
```

7.7 Building Login Authentication Systems

Let's create a secure login system that demonstrates string comparisons and decision-making:

Listing 29: Secure Login Authentication System

```
# Secure Login Authentication System
print("=" * 50)
print("SECURE LOGIN SYSTEM")
print("=" * 50)

# Define valid user credentials
valid_users = {
    "student": "python123",
    "admin": "admin456",
    "teacher": "education789",
    "guest": "welcome"
}

# Track login attempts
max_attempts = 3
attempts = 0

print("Welcome! Please log in to continue.")
print("Valid users: student, admin, teacher, guest")
print(f"You have {max_attempts} attempts.\n")

# Login loop
while attempts < max_attempts:
    username = input("Username: ").lower().strip()
    password = input("Password: ")

    attempts += 1

    # Check if username exists
    if username in valid_users:
        # Check if password is correct
        if password == valid_users[username]:
            print(f"\n[Success checkmark] LOGIN SUCCESSFUL!")
            print(f"Welcome back, {username.title()}!")

            # Different messages based on user type
            if username == "admin":
                print("[Wrench emoji] Administrator access granted.")
                print("You have full system privileges.")
            elif username == "teacher":
                print("[Books emoji] Teacher access granted.")
                print("You can manage courses and students.")
            elif username == "student":
                print("[Book emoji] Student access granted.")
                print("You can access course materials.")
            elif username == "guest":
                print("[Wave emoji] Guest access granted.")
                print("You have limited viewing privileges.")

            print(f"Login completed on attempt {attempts} of {max_attempts}")
            break

        else:
            remaining = max_attempts - attempts
            if remaining > 0:
                print(f"[X mark] Incorrect password for {username}")
                print(f"Attempts remaining: {remaining}\n")
            else:
                print(f"[X mark] Incorrect password for {username}")
    else:
        remaining = max_attempts - attempts
        if remaining > 0:
            print(f"[X mark] Username ’{username}’ not found")
            print(f"Attempts remaining: {remaining}\n")
        else:
            print(f"[X mark] Username ’{username}’ not found")

# Handle failed login
if attempts >= max_attempts:
    print("\n[Prohibited emoji] ACCOUNT LOCKED")
    print("Too many failed login attempts.")
    print("Please contact system administrator.")

print("\n" + "=" * 50)
print("Login session ended.")
```

8 Part V: Objects and Wrap-up

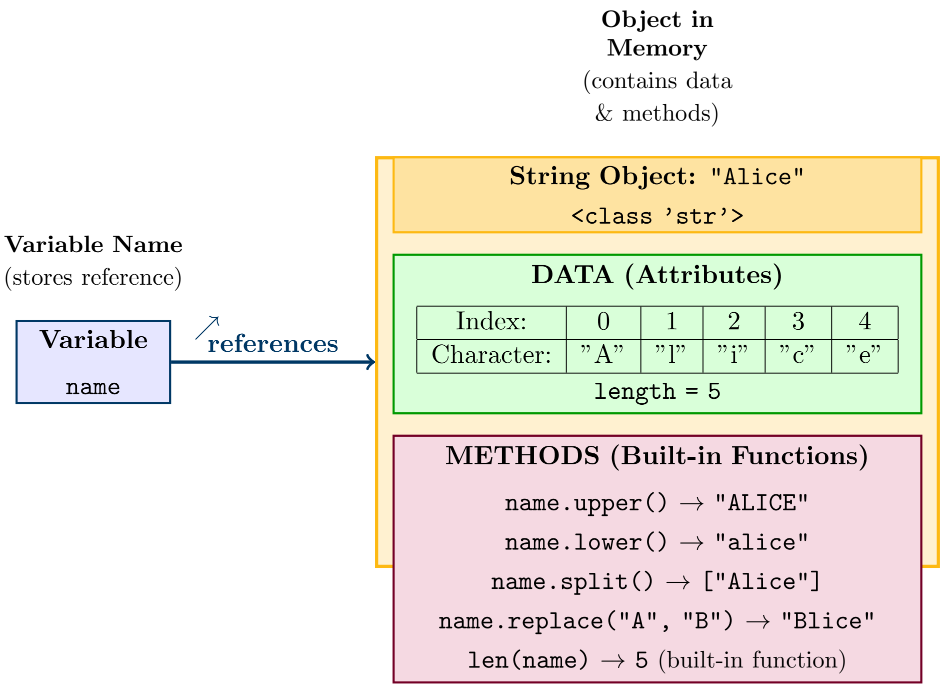
8.1 Python’s Object Model Explained

Understanding that ”everything in Python is an object” is fundamental to mastering the language. This concept explains many of Python’s behaviors and capabilities.

8.1.1 What Are Objects?

In Python, objects are instances of data types that contain both data (attributes) and functionality (methods):

- **Objects are instances** of a data type (class)
- **Every value has a type** and identity in memory
- **Objects have attributes** (data) and methods (functions)
- **Even simple values** like numbers and strings are objects



Index:	0	1	2	3	4
Character:	”A”	”l”	”i”	”c”	”e”

8.1.2 Examples of Python Objects

Listing 30: Everything is an Object in Python

```python
# Numbers are objects with methods
number = 42
print(f"Number: {number}")
print(f"Type: {type(number)}")
print(f"Methods available: {dir(number)[:5]}")  # Show first 5 methods

# Strings are objects with many useful methods
text = "Hello, World!"
print(f"\nString: {text}")
print(f"Type: {type(text)}")
print(f"Uppercase: {text.upper()}")
print(f"Lowercase: {text.lower()}")
print(f"Split into words: {text.split()}")
print(f"Replace ’World’ with ’Python’: {text.replace(’World’, ’Python’)}")

# Even functions are objects!
print(f"\nprint function type: {type(print)}")
print(f"len function type: {type(len)}")

# Lists are objects (preview of future topic)
my_list = [1, 2, 3]
print(f"\nList: {my_list}")
print(f"Type: {type(my_list)}")
my_list.append(4)  # Using a method to modify the list
print(f"After append: {my_list}")
```

8.2 Dynamic Typing in Python

Learning Objective: Understand how Python variables can hold different types of data and why this flexibility is powerful for programming.

8.2.1 What Does Dynamic Typing Mean?

Dynamic typing means that the same variable can hold different types of data throughout your program. Unlike some programming languages where you must declare what type of data a variable will hold, Python figures this out automatically.

Key Benefits:

- **Flexibility:** Variables can adapt to different data as your program runs
- **Simplicity:** No need to declare types in advance
- **Rapid Development:** Write code faster without worrying about type declarations

Real-World Analogy: Think of a variable like a box that can hold different things at different times - sometimes a number, sometimes text, sometimes a list of items.

8.2.2 Dynamic Typing in Action

Let's see how one variable can transform to meet different needs:

Listing 31: Dynamic Typing - Same Variable Multiple Types

```python
# Same variable name, different data types!
user_data = "Alice"              # Start with a string
print(f"Name: {user_data} (type: {type(user_data)})")

user_data = 25                   # Now it holds an integer age
print(f"Age: {user_data} (type: {type(user_data)})")

user_data = 85.5                 # Now it holds a float grade
print(f"Grade: {user_data} (type: {type(user_data)})")

user_data = True                 # Now it holds a boolean status
print(f"Enrolled: {user_data} (type: {type(user_data)})")
```

What Just Happened? The variable `user_data` successfully held four completely different types of information. Python automatically detected each type and handled the transitions seamlessly.

Practical Benefits in Real Programming:

- **User Input Processing:** Variables can start as strings from input(), then convert to numbers for calculations
- **Data Analysis:** Variables can hold different data structures as you process information
- **Error Handling:** Variables can hold error messages (strings) or valid results (numbers)
- **Flexibility:** Change what your program does without rewriting variable declarations

8.2.3 Understanding Variable References

Key Concept: When you assign a value to a variable in Python, you're creating a reference to an object in memory, not copying the value itself.

Why This Matters: Understanding references helps you predict how Python behaves and avoid common programming mistakes.

Listing 32: Variable References Explained

```python
# Two variables can point to the same object
original_score = 95
final_score = original_score    # Both variables reference the same object

print(f"Original: {original_score}")
print(f"Final: {final_score}")
print(f"Same object? {original_score is final_score}")  # True

# Changing the reference doesn't affect the original
final_score = 98                # final_score now references a different object
print(f"\nAfter change:")
print(f"Original: {original_score}")  # Still 95!
print(f"Final: {final_score}")        # Now 98
print(f"Same object? {original_score is final_score}")  # False
```

Practical Applications:

- **Memory Efficiency:** Python reuses objects when possible to save memory
- **Comparison Operations:** Understanding the difference between `==` (same value) and `is` (same object)
- **Function Parameters:** How variables are passed to functions (by reference)

Simple Rule: For basic data types (numbers, strings, booleans), you rarely need to worry about references. Python handles the details automatically!

8.3 Preview: Descriptive Statistics with Python

Now let's see how today's fundamental concepts connect to real data science applications. We'll explore this step-by-step to show how variables, arithmetic, and decision-making combine to solve analytical problems.

8.3.1 What Are Descriptive Statistics?

Definition: Descriptive statistics are numbers that summarize and describe the important features of a dataset. They help us understand what our data tells us.

The Three Essential Statistics:

- **Mean (Average):** The typical value - calculated by adding all values and dividing by the count
- **Range:** The spread - difference between the highest and lowest values
- **Minimum/Maximum:** The extreme values in our dataset

8.3.2 Simple Example: Understanding the Mean

Let's start with a simple example using today's arithmetic and variable concepts:

Listing 33: Calculating a Mean - Step by Step

```python
# Step 1: Store data in variables (using today's variable skills)
quiz1 = 85
quiz2 = 92
```

```python
quiz3 = 78

# Step 2: Calculate total using arithmetic from today
total_points = quiz1 + quiz2 + quiz3
print(f"Total points: {total_points}")

# Step 3: Calculate mean using division
num_quizzes = 3
mean_score = total_points / num_quizzes
print(f"Mean (average) score: {mean_score:.1f}")

# Step 4: Find range using today's comparison concepts
highest = max(quiz1, quiz2, quiz3)
lowest = min(quiz1, quiz2, quiz3)
score_range = highest - lowest
print(f"Range: {score_range} points")
```

8.3.3 Practical Application: Student Grade Analysis

Now let's build a complete grade analyzer using today's concepts:

Listing 34: Grade Analyzer Using Today's Skills

```python
# Get student data using input() from today
student_name = input("Enter student name: ")
grade1 = float(input("Enter first grade: "))
grade2 = float(input("Enter second grade: "))
grade3 = float(input("Enter third grade: "))

# Calculate statistics using today's arithmetic
total = grade1 + grade2 + grade3
average = total / 3
highest = max(grade1, grade2, grade3)
lowest = min(grade1, grade2, grade3)

# Display results using f-strings from today
print(f"\n{student_name}'s Grade Summary:")
print(f"Average: {average:.1f}")
print(f"Highest: {highest}")
print(f"Lowest: {lowest}")

# Make decisions using if statements from today
if average >= 90:
    grade_letter = "A"
elif average >= 80:
    grade_letter = "B"
elif average >= 70:
    grade_letter = "C"
else:
    grade_letter = "Needs Improvement"

print(f"Overall Performance: {grade_letter}")
```

8.3.4 Connection to Today's Learning

Notice how this statistical application directly uses every major concept from today's lecture:

- **Variables:** Store individual grades and calculated results

- **Arithmetic Operations:** Addition for totals, division for averages
- **Input/Output:** Get user data and display professional results
- **Decision Making:** Assign letter grades based on numerical scores
- **Type Conversion:** Convert input strings to numbers for calculations

Key Insight: Complex data science applications are built from the same fundamental building blocks you learned today!

8.3.5 Python Tools for Future Statistical Work

Built-in Functions We'll Master:

- `sum()` - Calculate totals
- `len()` - Count items
- `min()` and `max()` - Find extremes
- `sorted()` - Arrange data in order

Libraries We'll Learn:

- **statistics module:** mean(), median(), mode(), stdev()
- **NumPy:** Advanced numerical computing with arrays
- **pandas:** Data manipulation and analysis
- **matplotlib:** Creating professional charts and graphs

Real-World Applications:

- Analyzing survey data and customer feedback
- Financial analysis and investment tracking
- Sports statistics and performance metrics
- Scientific research data processing
- Business intelligence and reporting

9 Today's Accomplishments

Let's review everything you've mastered in today's comprehensive Python programming journey:

9.1 Part I: Variables and Assignment ✓

Skills Mastered:

- Created variables with meaningful, descriptive names
- Worked confidently with all data types (int, float, str, bool)
- Used the type() function to verify and debug data types
- Applied Python naming conventions (PEP 8) professionally
- Built a complete personal information system

9.2 Part II: Arithmetic Operations ✓

Skills Mastered:

- Used all seven arithmetic operators (+, -, *, **, /, //, %) correctly
- Applied operator precedence rules (PEMDAS) to complex expressions
- Implemented augmented assignment operators (+=, -=, etc.) efficiently
- Built sophisticated financial calculators (compound interest, mortgage)
- Solved real-world mathematical problems with code

9.3 Part III: Input/Output and Strings ✓

Skills Mastered:

- Mastered advanced print() function features with custom separators and endings
- Created professional output using powerful f-string formatting
- Got and processed user input with proper type conversion
- Built interactive programs that respond to user needs
- Developed a complete greeting system and calculator

9.4 Part IV: Decision Making with if ✓

Skills Mastered:

- Used boolean values and comparison operators for logical decisions
- Implemented if statements with proper Python indentation
- Created interactive programs with user authentication
- Performed string comparisons and input validation
- Built comprehensive age verification and login systems

9.5 Part V: Objects and Python Foundation ✓

Skills Mastered:

- Understood Python's object model and dynamic typing
- Explored object identity and references in memory
- Previewed statistical applications of programming concepts
- Connected today's fundamentals to future data science work

10 Interactive Programs Built Today

Seven Complete Programs You Created:

1. **Personal Data Variables System** - Comprehensive information storage and display
2. **Compound Interest Calculator** - Financial mathematics with real formulas
3. **Restaurant Bill Calculator** - Complex arithmetic with tax and tip calculations
4. **Interactive Greeting Generator** - Advanced input processing and personalization
5. **Python Calculator** - Full-featured calculator with error handling
6. **Age Verification System** - Sophisticated decision-making with multiple conditions
7. **Login Authentication System** - Secure user authentication with validation

Programming Concepts Applied:

- Variable creation, manipulation, and validation
- User input processing and type conversion
- Mathematical calculations and financial modeling
- Conditional logic and decision trees
- String processing and comparison techniques
- Professional output formatting and user experience
- Error handling and input validation
- Program structure and organization

11 Next Steps in Your Python Journey

11.1 Building on Today's Foundation

Today's five-part journey provides the solid foundation for everything we'll build in this course. You now understand:

- How to think like a programmer and break problems into steps
- The core building blocks of Python: variables, arithmetic, input/output, and decisions
- How to write professional, readable code that solves real-world problems
- The object-oriented nature of Python and its practical implications
- How basic programming concepts connect to advanced data science applications

11.2 Coming in Future Lectures

Lecture 3: Functions and Error Handling

- Writing reusable functions with parameters and return values
- Understanding scope and local vs global variables
- Handling errors gracefully with try/except blocks
- Code organization and modularity principles
- Building function libraries for common tasks

Lecture 4: Lists and Data Structures

- Creating and manipulating lists of data
- Working with tuples and dictionaries
- Iterating through data with loops
- Processing collections of information
- Building database-like systems

11.3 Practice Opportunities

Immediate Extensions (Try This Week):

- Extend today's calculator with more operations (square root, percentage)
- Create a personal budget calculator with income and expenses
- Build a quiz program with multiple questions and scoring
- Add more validation to the login system (password strength, account lockout)
- Experiment with different f-string formatting options

Challenge Projects (For Advanced Practice):

- Grade point average calculator with weighted courses
- Investment portfolio tracker with multiple stocks
- Temperature converter with multiple units (Celsius, Fahrenheit, Kelvin)
- Simple encryption/decryption program using ASCII values
- Text-based adventure game with player choices

12 Study Recommendations

Review Materials:

- Re-read Deitel Chapter 2 (pages 49-72) with today's hands-on experience
- Practice all seven interactive exercises from today's handout
- Work through the textbook's self-review exercises and answers
- Experiment with variations of today's programs

Hands-On Practice:

- Create your own versions of today's programs with personal data
- Try different input validation scenarios
- Experiment with edge cases (What happens with negative numbers? Empty strings?)
- Practice debugging by intentionally introducing errors and fixing them

Conceptual Understanding:

- Make sure you understand WHY each concept is important
- Connect programming concepts to real-world applications
- Practice explaining concepts to others (teaching reinforces learning)
- Ask yourself: "How could I use this in my field of study?"

13 Key Takeaways and Summary

Today's Transformation:

You started today with basic Python knowledge and now have the core skills to build meaningful, interactive programs. The five-part journey we completed represents the essential foundation every Python programmer must master.

Essential Programming Skills You Now Possess:

- **Problem Decomposition:** Breaking complex tasks into manageable steps
- **Data Management:** Creating and manipulating variables of different types
- **Mathematical Computing:** Performing calculations with proper precedence
- **User Interaction:** Creating programs that respond to user input
- **Decision Logic:** Building programs that make intelligent choices
- **Code Organization:** Writing readable, maintainable code with good style
- **Professional Output:** Creating polished, user-friendly interfaces

The Foundation is Set: Variables, arithmetic, input/output, and decisions give you the tools to build programs that solve real-world problems. From here, we'll add more sophisticated features, but everything builds on what you learned today.

Remember: Programming is learned by doing - keep coding, keep experimenting, and keep building!

Congratulations on Your Python Programming Achievement!

You've built a solid foundation in Python programming. Every expert programmer started exactly where you are now. The key is persistence, practice, and curiosity about what's possible!

See you next class for Functions and Error Handling!

Programming is a journey of continuous learning and growth. Today you took significant steps forward - tomorrow we'll build even more powerful and sophisticated programs!