Lecture 1: Introduction to Computers and Python

Course: INF 605 - Introduction to Programming - Python
Instructor: Prof. Rongyu Lin
Institution: Quinnipiac University

Learning Objectives

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

1. Explain the relationship between hardware, software, and programming
2. Write and execute your first Python program
3. Use Python as a powerful calculator with arithmetic operators
4. Create variables to store different data types
5. Work with strings and format output using f-strings
6. Use basic math library functions

Prerequisites

No prior programming experience required. This is your first step into the world of Python programming.

Setup and Environment Check

Before we begin coding, let's verify that your Python environment is working correctly. Every programmer starts by ensuring their tools are ready. Think of this like a pilot checking their instruments before takeoff - we want to make sure everything is functioning properly before we start our programming journey.

In [1]:
```
# Verify Python environment
import sys
print(f"Python version: {sys.version.split()[0]}")
print("Your environment is ready for programming!")
```

What Happened: We imported the `sys` module and used it to display your Python version. If you see a version number (like 3.11 or 3.12), your environment is working correctly. The `print()` function displays text on the screen - you'll use this constantly in Python.

Part 1: Your First Python Program

The Famous "Hello, World!" Program

Every programmer's journey begins with the classic "Hello, World!" program. This tradition dates back to 1978 and serves as a simple way to verify that you can write, run, and see the output of a program. In Python, this program is remarkably simple - just one line of code. The `print()` function is one of the most fundamental tools in Python, allowing you to display text, numbers, and other data on the screen.

In [1]:
```
# Your first Python program
print("Hello, World!")
```

What Happened: The `print()` function took the text inside the parentheses and displayed it on the screen. The text is enclosed in quotation marks because it's a "string" - Python's way of representing text. You can use either single quotes `'Hello'` or double quotes `"Hello"` - both work exactly the same way.

Printing Multiple Lines

Programs rarely display just one line of text. Let's explore how to print multiple messages. Each `print()` statement automatically moves to a new line after displaying its content. This is how we build up more complex output, one line at a time.

In [1]:
```
# Printing multiple lines
print("Welcome to Python Programming!")
print("This is INF 605.")
print("Let's learn together!")
```

What Happened: Each `print()` statement displayed its text and then moved to a new line. Python executes these statements in order, from top to bottom. This sequential execution is fundamental to how programs work - instructions are followed one at a time, in the order they appear.

Print with Multiple Arguments

The `print()` function can accept multiple items separated by commas. When you provide multiple arguments, Python automatically inserts a space between each item when displaying them. This is a convenient way to combine different pieces of information in a single line of output.

In [1]:
```
# Print with multiple arguments
print("Python", "is", "awesome!")
print("2", "+", "2", "=", 4)
```

What Happened: When we pass multiple items to `print()`, Python displays them with spaces in between. Notice that the number `4` doesn't have quotes around it - that's because it's a number, not text. Python treats numbers and text differently, and we'll explore this distinction more later.

Exercise 1: Your Personal Greeting

Write three print statements:

1. Print your name
2. Print your major
3. Print why you're interested in learning Python

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
print("My name is Alice Johnson")
print("I am majoring in Computer Science")
print("I want to learn Python for data analysis")
```

Part 2: Python as a Calculator

Basic Arithmetic Operations

One of Python's most immediately useful features is its ability to perform mathematical calculations. You can use Python just like a calculator, but much more powerful. Python supports all the basic arithmetic operations you learned in math class: addition, subtraction, multiplication, and division. Let's start with the simplest operations.

Addition and Subtraction

Addition uses the `+` symbol and subtraction uses the `-` symbol, just like you would write on paper. Python can handle whole numbers (integers) and decimal numbers (floats) in these operations. The results are calculated instantly and can be displayed using `print()`.

In [1]:
```
# Addition examples
print("Addition:")
print("5 + 3 =", 5 + 3)
print("100 + 250 =", 100 + 250)
print("1.5 + 2.7 =", 1.5 + 2.7)
```

What Happened: Python calculated each sum and displayed the result. Notice that `1.5 + 2.7` gives us `4.2` - Python handles decimal numbers seamlessly. The calculation happens first, then `print()` displays both the equation and its result.

In [1]:
```
# Subtraction examples
print("Subtraction:")
print("10 - 4 =", 10 - 4)
print("50 - 23 =", 50 - 23)
print("7.8 - 3.2 =", 7.8 - 3.2)
```

What Happened: Subtraction works exactly as expected. Python can also handle negative results - try `5 - 10` and you'll get `-5`. This ability to work with negative numbers is essential for many real-world calculations.

Multiplication and Division

Multiplication uses the asterisk `*` symbol (not `x` as in math class). Division uses the forward slash `/` symbol. One important thing to note: in Python 3, regular division always returns a decimal number, even when dividing two whole numbers that divide evenly.

In [1]:
```
# Multiplication examples
print("Multiplication:")
print("6 * 7 =", 6 * 7)
print("8 * 9 =", 8 * 9)
print("3.14 * 2 =", 3.14 * 2)
```

What Happened: The `*` operator multiplies numbers together. This is the same multiplication you know from math, just written with `*` instead of the multiplication sign. Python handles both integers and decimals smoothly.

```
In [1]: # Division examples
        print("Division:")
        print("15 / 3 =", 15 / 3)
        print("17 / 4 =", 17 / 4)
        print("22 / 7 =", 22 / 7)
```

What Happened: Notice that `15 / 3` gives `5.0`, not `5`. In Python 3, the `/` operator always returns a decimal (float) result. This is actually helpful because it ensures precision in calculations. The result `22 / 7` gives an approximation of pi.

Floor Division and Modulus

Python provides two special division operators that are incredibly useful. Floor division `//` gives you just the whole number part of a division (rounding down). The modulus operator `%` gives you the remainder after division. These operators are used constantly in programming for tasks like determining if a number is even or odd, or converting units.

```
In [1]: # Floor division - gives whole number part only
        print("Floor Division (//):")
        print("17 // 4 =", 17 // 4)
        print("22 // 7 =", 22 // 7)
        print("15 // 3 =", 15 // 3)
```

What Happened: Floor division discards the decimal part entirely. When we calculate `17 // 4`, Python determines that 4 goes into 17 exactly 4 times (with a remainder), so it returns `4`. This is useful when you need a whole number result, like calculating how many complete boxes you can fill.

Exercise 2: Regular Division vs Floor Division

Regular division `/` always gives a decimal. Floor division `//` keeps only the whole-number part (rounds down).

Calculate and print each pair. Then say which result you would use in real life.

1. `20 / 6` and `20 // 6`
2. `9 / 2` and `9 // 2`
3. You have 23 snacks and boxes that hold 4 snacks each. Print how many **full boxes** you can fill (`//`) and the exact share if 23 snacks are split among 4 people (`/`).
4. Print `8 / 2` and `8 // 2` , then print the type of each result (`type(...)`). Are they the same type?

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
print("1. Regular vs floor:")
print("20 / 6 =", 20 / 6)
print("20 // 6 =", 20 // 6)

print("2. Regular vs floor:")
print("9 / 2 =", 9 / 2)
print("9 // 2 =", 9 // 2)

print("3. Snacks and boxes:")
print("Full boxes:", 23 // 4)
print("Exact share among 4 people:", 23 / 4)

print("4. Types:")
print("8 / 2 =", 8 / 2, "type:", type(8 / 2))
print("8 // 2 =", 8 // 2, "type:", type(8 // 2))
print("They are not the same type: / gives a float, // gives
an int.")
```

In [1]:
```
# Modulus - gives the remainder
print("Modulus (%):")
print("17 % 4 =", 17 % 4)
print("20 % 5 =", 20 % 5)
print("23 % 7 =", 23 % 7)
```

What Happened: The modulus operator returns what's left over after division. For `17 % 4` : 4 goes into 17 four times (4 x 4 = 16), leaving a remainder of 1. When the result is `0` (like `20 % 5`), it means the first number divides evenly by the second.

Exponentiation

To raise a number to a power, Python uses the double asterisk `**` operator. This is called exponentiation. For example, `2 ** 3` means "2 raised to the power of 3" or "2 cubed", which equals 8. This operator is essential for scientific calculations, financial formulas, and many other applications.

In [1]:
```
# Exponentiation - raising to powers
print("Exponentiation (**):")
print("2 ** 3 =", 2 ** 3)
print("5 ** 2 =", 5 ** 2)
print("10 ** 6 =", 10 ** 6)
```

What Happened: The `**` operator calculates powers. `2 ** 3` is 2 multiplied by itself 3 times (2 x 2 x 2 = 8). `5 ** 2` is 5 squared (25). `10 ** 6` is one million - notice how quickly powers can grow to large numbers!

In [1]:
```
# Square roots using fractional exponents
print("Square Roots:")
print("9 ** 0.5 =", 9 ** 0.5)
print("16 ** 0.5 =", 16 ** 0.5)
print("25 ** 0.5 =", 25 ** 0.5)
```

What Happened: A clever trick: raising a number to the power of 0.5 calculates its square root. This works because the square root of a number is the same as raising it to the power of 1/2. The square root of 9 is 3, because 3 x 3 = 9.

Order of Operations (PEMDAS)

Python follows the standard mathematical order of operations, often remembered as PEMDAS: Parentheses, Exponents, Multiplication/Division (left to right), Addition/Subtraction (left to right). Understanding this order is crucial for writing correct calculations. When in doubt, use parentheses to make your intentions explicit.

In [1]:
```
# Order of operations examples
print("Order of Operations:")
print("2 + 3 * 4 =", 2 + 3 * 4)
print("(2 + 3) * 4 =", (2 + 3) * 4)
```

What Happened: In `2 + 3 * 4`, multiplication happens first (3 * 4 = 12), then addition (2 + 12 = 14). In `(2 + 3) * 4`, parentheses force addition first (2 + 3 = 5), then multiplication (5 * 4 = 20). The parentheses completely change the result!

In [1]:
```
# More complex examples
print("Complex Calculations:")
print("2 ** 3 * 4 =", 2 ** 3 * 4)
print("3 ** 2 + 4 ** 2 =", 3 ** 2 + 4 ** 2)
```

What Happened: Exponents are calculated before multiplication. In `2 ** 3 * 4`: first `2 ** 3 = 8`, then `8 * 4 = 32`. The second example calculates the Pythagorean theorem: `3**2 + 4**2 = 9 + 16 = 25`. The square root of 25 is 5, completing the famous 3-4-5 right triangle.

Exercise 3: Calculator Practice

Calculate the following:

1. The area of a rectangle with width 15 and height 8
2. The remainder when 100 is divided by 7
3. 2 raised to the power of 10

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
print("Rectangle area:", 15 * 8)
print("100 divided by 7, remainder:", 100 % 7)
print("2 to the power of 10:", 2 ** 10)
```

Part 3: Variables

What Are Variables?

Variables are like labeled containers that store values in your computer's memory. Instead of using raw numbers everywhere, you can give them meaningful names. This makes your code easier to read, understand, and modify. Think of variables as sticky notes - you write a name on the note and attach it to a value, so you can refer to that value later by its name.

In [1]:
```
# Creating variables
student_name = "Alice"
student_age = 20

print("Name:", student_name)
print("Age:", student_age)
```

What Happened: We created two variables: `student_name` stores the text "Alice" and `student_age` stores the number 20. The equals sign `=` is the assignment operator - it takes the value on the right and stores it in the variable on the left. Once stored, we can use the variable name to access its value.

Using Variables in Calculations

Variables become powerful when you use them in calculations. Instead of hard-coding numbers, you can use variable names. This makes your code flexible - change the variable's value once, and all calculations using that variable automatically update. This is one of the fundamental concepts that makes programming so powerful.

In [1]:
```
# Variables in calculations
price = 19.99
quantity = 3
total = price * quantity

print("Price per item: $", price)
print("Quantity:", quantity)
print("Total: $", total)
```

What Happened: We stored the price and quantity in variables, then calculated the total using those variables. The calculation `price * quantity` uses the values stored in the variables (19.99 and 3) to compute 59.97. If you change the price or quantity, the total will automatically reflect the new values when you run the code again.

Variable Naming Rules

Python has specific rules for naming variables. Names must start with a letter or underscore, can contain letters, numbers, and underscores, and are case-sensitive (`age` and `Age` are different variables). Good variable names are descriptive and follow the snake_case convention - lowercase words separated by underscores.

In [1]:
```
# Good variable names (descriptive and clear)
student_count = 25
average_score = 87.5
is_enrolled = True

print("Students:", student_count)
print("Average:", average_score)
print("Enrolled:", is_enrolled)
```

What Happened: These variable names follow Python conventions - they're lowercase, use underscores to separate words, and clearly describe what they contain. Good naming is crucial for code readability. When you or someone else reads this code months later, the purpose of each variable is immediately clear.

Updating Variables

Variables can be updated - their values can change as your program runs. This is essential for tracking changing information like scores, counters, or running totals. You can also update a variable based on its current value, which is extremely common in programming.

In [1]:
```
# Updating variables
score = 0
print("Initial score:", score)

score = score + 10
print("After bonus:", score)

score = score + 5
print("After another bonus:", score)
```

What Happened: We started with `score = 0`, then updated it twice. The expression `score = score + 10` means "take the current value of score (0), add 10 to it, and store the result (10) back in score." This pattern of updating a variable based on its current value is fundamental to programming.

Part 4: Data Types

Understanding Data Types

Python automatically determines the type of data you're working with. The four main data types you'll use constantly are: integers (whole numbers), floats (decimal numbers), strings (text), and booleans (True/False values). Understanding data types helps you predict how Python will behave with different kinds of information.

Integers (int)

Integers are whole numbers without decimal points. They can be positive, negative, or zero. Python can handle integers of any size - there's no practical limit to how big they can be. Integers are used for counting, indexing, and any situation where you need exact whole numbers.

In [1]:
```
# Integer examples
age = 25
temperature = -5
year = 2025

print("Age:", age, "- Type:", type(age))
print("Temperature:", temperature, "- Type:",
type(temperature))
print("Year:", year, "- Type:", type(year))
```

What Happened: The `type()` function reveals what kind of data Python is storing. All three variables are `<class 'int'>` - integers. Notice that negative numbers and large numbers like 2025 are both integers. Python treats them all the same way for calculations.

Floats (float)

Floats are numbers with decimal points. They're used for measurements, percentages, currency, and any calculation requiring precision beyond whole numbers. The name "float" comes from "floating-point number," which refers to how computers store decimal values internally.

In [1]:
```
# Float examples
height = 5.9
price = 19.99
pi = 3.14159

print("Height:", height, "- Type:", type(height))
print("Price:", price, "- Type:", type(price))
print("Pi:", pi, "- Type:", type(pi))
```

What Happened: These are all `<class 'float'>` - floating-point numbers. Even a number like `5.0` (which equals 5) would be a float because it has a decimal point. Python keeps track of this distinction because integers and floats are stored differently in memory.

Strings (str)

Strings are sequences of characters - text data. They must be enclosed in quotes (single or double). Strings can contain letters, numbers, symbols, and even spaces. Unlike numbers, strings cannot be used directly in mathematical calculations, but they have their own powerful operations.

In [1]:
```
# String examples
first_name = "Alice"
last_name = 'Johnson'
message = "Hello, World!"

print("First:", first_name, "- Type:", type(first_name))
print("Last:", last_name, "- Type:", type(last_name))
print("Message:", message, "- Type:", type(message))
```

What Happened: All three are `<class 'str'>` - strings. Notice that both single quotes (`'Johnson'`) and double quotes (`"Alice"`) work identically. The choice is often based on what the string contains - if your text has an apostrophe, use double quotes to avoid confusion.

Booleans (bool)

Booleans represent one of two values: `True` or `False` . They're named after mathematician George Boole. Booleans are essential for making decisions in programs - they answer yes/no questions. Note that `True` and `False` must be capitalized exactly this way in Python.

In [1]:
```
# Boolean examples
is_student = True
is_graduated = False

print("Is student:", is_student, "- Type:",
type(is_student))
print("Is graduated:", is_graduated, "- Type:",
type(is_graduated))
```

What Happened: Both variables are `<class 'bool'>` - booleans. Booleans often result from comparisons. For example, `5 > 3` evaluates to `True` because 5 is indeed greater than 3. We'll use booleans extensively when we learn about decision-making in the next lecture.

Exercise 4: Data Types

Create variables for the following and print each with its type:

1. Your birth year (integer)
2. Your GPA (float)
3. Your favorite color (string)
4. Whether you like programming (boolean)

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
birth_year = 2003
gpa = 3.75
favorite_color = "blue"
likes_programming = True

print("Birth year:", birth_year, "- Type:",
type(birth_year))
print("GPA:", gpa, "- Type:", type(gpa))
print("Favorite color:", favorite_color, "- Type:",
type(favorite_color))
print("Likes programming:", likes_programming, "- Type:",
type(likes_programming))
```

Part 5: Working with Strings

String Concatenation

String concatenation means joining strings together. In Python, you use the `+` operator to combine strings. This is different from adding numbers - when you use `+` with strings, Python glues them together end-to-end rather than performing mathematical addition.

In [1]:
```
# String concatenation
first_name = "Alice"
last_name = "Johnson"

full_name = first_name + " " + last_name
full_name  # Jupyter displays: 'Alice Johnson'
```

What Happened: We joined three strings: "Alice", a space " ", and "Johnson" to create "Alice Johnson". Notice we had to explicitly add the space - Python doesn't add spaces automatically when concatenating. This gives you complete control over the final result.

String Repetition

You can repeat a string multiple times using the `*` operator with a number. This is useful for creating separators, patterns, or repeated text. The number tells Python how many times to repeat the string.

In [1]:
```
# String repetition
separator = "=" * 30
print(separator)
print("Welcome!")
print(separator)
```

What Happened: The expression `"=" * 30` creates a string of 30 equal signs. This is a common technique for creating visual separators in program output. You could use any character - dashes, asterisks, or any other symbol.

String Length

The `len()` function returns the number of characters in a string, including spaces and punctuation. This is useful for validation (checking if input is too long or too short) and many text processing tasks.

In [1]:
```
# String length
message = "Hello, World!"

# len() returns the number of characters
len(message)  # Jupyter displays: 13
```

What Happened: The string "Hello, World!" contains 13 characters - including the comma, space, and exclamation mark. The `len()` function counts every character, not just letters. An empty string `""` would have a length of 0.

String Methods

Strings have built-in methods - special functions that work on string data. Methods are called using dot notation: `string.method()`. Common methods include `upper()` (convert to uppercase), `lower()` (convert to lowercase), and `replace()` (substitute text).

In [1]:
```
# String methods
text = "Python Programming"

print("Original:", text)
print("Uppercase:", text.upper())
print("Lowercase:", text.lower())
```

What Happened: The `upper()` method returns a new string with all characters converted to uppercase. The `lower()` method does the opposite. Note that these methods don't change the original string - they create and return a new string with the modification.

In [1]:
```
# Replace method
text = "I love Java"
new_text = text.replace("Java", "Python")

print("Original:", text)
print("After replace:", new_text)
```

What Happened: The `replace()` method takes two arguments: the text to find and the text to substitute. It searches for all occurrences of the first string and replaces them with the second. The original string remains unchanged - `replace()` returns a new modified string.

Part 6: F-Strings (Formatted String Literals)

What Are F-Strings?

F-strings are Python's most elegant way to include variables and expressions inside strings. Add an `f` before the opening quote, then use curly braces `{}` to insert variables or calculations. F-strings make your code more readable and are generally preferred over older formatting methods.

In [1]:
```
# Basic f-string usage
name = "Alice"
age = 20

print(f"My name is {name} and I am {age} years old.")
```

What Happened: The `f` before the quote marks this as a formatted string. Python replaces `{name}` with the value "Alice" and `{age}` with 20. This is much cleaner than concatenating strings with `+` operators and easier to read than older formatting methods.

Calculations in F-Strings

F-strings can include expressions, not just variables. You can perform calculations directly inside the curly braces. Python evaluates the expression and inserts the result into the string. This is incredibly convenient for displaying calculated values.

In [1]:
```
# Calculations in f-strings
price = 19.99
quantity = 3

print(f"Total cost: ${price * quantity}")
print(f"In 5 years, I will be {age + 5} years old.")
```

What Happened: Inside the curly braces, we placed expressions (`price * quantity` and `age + 5`). Python calculates these values and inserts the results into the string. You don't need to create separate variables for these calculations if you only need them for display.

Formatting Numbers

F-strings support format specifiers that control how numbers appear. The most common is `:.2f` which formats a number with exactly 2 decimal places. This is essential for displaying currency, percentages, and other formatted numbers.

In [1]:
```
# Formatting numbers in f-strings
price = 1234.5678

print(f"Default: {price}")
print(f"Two decimals: {price:.2f}")
print(f"With comma: {price:,.2f}")
```

What Happened: The format specifier comes after a colon inside the braces. `:.2f` means "format as float with 2 decimal places." Adding a comma (`:,.2f`) adds thousand separators. These formatting options make your output professional and readable.

Exercise 5: F-String Practice

Create variables for a product name, price, and quantity. Then use f-strings to print:

1. "Product: [name]"
2. "Price: $[price] (with 2 decimal places)"
3. "Quantity: [quantity]"
4. "Total: $[calculated total with 2 decimal places]"

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
product = "Laptop"
price = 899.99
quantity = 2

print(f"Product: {product}")
print(f"Price: ${price:.2f}")
print(f"Quantity: {quantity}")
print(f"Total: ${price * quantity:.2f}")
```

Part 7: The Math Library

Importing the Math Module

Python's math library provides advanced mathematical functions and constants that aren't available as basic operators. To use it, you first import it with `import math`. Once imported, you access its functions using `math.function_name()`. The math library includes constants like pi and functions like square root.

In [1]:
```
# Import the math library
import math

# Mathematical constants
print("Mathematical Constants:")
print(f"Pi: {math.pi}")
print(f"e: {math.e}")
```

What Happened: We imported the math module, giving us access to mathematical constants. `math.pi` is the ratio of a circle's circumference to its diameter (approximately 3.14159). `math.e` is Euler's number (approximately 2.71828), the base of natural logarithms.

Square Root Function

The `math.sqrt()` function calculates the square root of a number. While we can also use `** 0.5` for square roots, `math.sqrt()` is more readable and clearly communicates your intent to other programmers.

In [1]:
```
# Square root examples
print("Square Roots:")
print(f"Square root of 16: {math.sqrt(16)}")
print(f"Square root of 2: {math.sqrt(2):.4f}")
print(f"Square root of 100: {math.sqrt(100)}")
```

What Happened: `math.sqrt()` returns the square root of the given number. The square root of 16 is 4 (because 4 x 4 = 16). The square root of 2 is an irrational number, so we formatted it to 4 decimal places for readability.

Power and Rounding Functions

The math library provides `pow()` for powers, `ceil()` for rounding up, and `floor()` for rounding down. While Python has built-in `**` for powers and `round()` for rounding, the math versions offer some additional capabilities.

In [1]:
```
# Power function
print("Powers:")
print(f"2 to the power of 10: {math.pow(2, 10)}")
```

In [1]:
```
# Rounding functions
print("Rounding:")
print(f"Ceiling of 3.2: {math.ceil(3.2)}")
print(f"Floor of 3.8: {math.floor(3.8)}")
```

What Happened: `math.ceil()` rounds UP to the nearest integer (ceiling means "above"). `math.floor()` rounds DOWN to the nearest integer (floor means "below"). These are different from regular rounding - `ceil(3.2)` gives 4 (not 3), and `floor(3.8)` gives 3 (not 4).

Practical Example: Circle Calculations

Let's use what we've learned to calculate properties of a circle. This combines variables, f-strings, math operations, and the math library into a practical application.

In [1]:
```
# Circle calculations
radius = 5

diameter = 2 * radius
circumference = 2 * math.pi * radius
area = math.pi * radius ** 2

print("Circle Calculator")
print("=" * 25)
print(f"Radius: {radius}")
print(f"Diameter: {diameter}")
print(f"Circumference: {circumference:.2f}")
print(f"Area: {area:.2f}")
```

What Happened: We calculated all properties of a circle with radius 5. The circumference formula is 2 x pi x r, and the area formula is pi x r squared. Using `math.pi` gives us an accurate value of pi. The `.2f` formatting makes the output clean and professional.

Exercise 6: Temperature Converter

Write a program that:

1. Stores a temperature in Celsius (try 25)
2. Converts it to Fahrenheit using the formula: F = (C * 9/5) + 32
3. Prints both temperatures formatted to 1 decimal place

In [1]:
```
# Your code here
```

In [1]:
```
# Solution
celsius = 25
fahrenheit = (celsius * 9/5) + 32

print("Temperature Converter")
print("=" * 25)
print(f"Celsius: {celsius:.1f} C")
print(f"Fahrenheit: {fahrenheit:.1f} F")
```

Part 8: Python Indentation

Why Indentation Matters

Python uses indentation (spaces at the beginning of lines) to define code structure. Unlike other languages that use braces {} or keywords, Python relies on consistent indentation. This forces code to be visually organized and readable. Standard practice is 4 spaces per indentation level. We'll use indentation extensively starting in Lecture 2 with if statements and loops.

In [1]:
```
# Preview: indentation with an if statement
age = 20

if age >= 18:
    print("You are an adult.")   # This line is indented
    print("You can vote.")       # This line is also
indented

print("This line always runs.")  # Not indented
```

What Happened: The two indented lines after `if age >= 18:` only run when the condition is true. The final line has no indentation, so it runs regardless of the condition. The colon `:` signals that indented code follows. We'll explore this fully in Lecture 2.

```
Visual Guide to Indentation:

if condition:
    |--- 4 spaces ---> indented code runs when condition is
True
    |--- 4 spaces ---> still part of the if block

not indented code always runs
```

Comprehensive Exercises

These two problems pull together print, arithmetic (including `/` and `//`), variables, types, and f-strings.

Comprehensive Exercise 1: Cafe Receipt

A cafe sells tea for $3.50 each. A student buys 7 teas and pays with a $50 bill.

1. Store the unit price, quantity, and cash paid in variables.
2. Compute the subtotal, 8% tax (`subtotal * 0.08`), and the total.
3. Compute the change.
4. How many **whole** $5 bills are in the change (`//`), and what is left after those fives (`%`)?
5. Use f-strings to print a short receipt (item, quantity, subtotal, tax, total, change) with money shown to two decimals (`:.2f`).

```
In [1]: # Your code here
```

In [1]:
```
# Solution
unit_price = 3.50
quantity = 7
cash_paid = 50

subtotal = unit_price * quantity
tax = subtotal * 0.08
total = subtotal + tax
change = cash_paid - total
five_bills = change // 5
leftover = change % 5

print("Cafe Receipt")
print("=" * 28)
print(f"Item: Tea")
print(f"Quantity: {quantity}")
print(f"Subtotal: ${subtotal:.2f}")
print(f"Tax (8%): ${tax:.2f}")
print(f"Total: ${total:.2f}")
print(f"Paid: ${cash_paid:.2f}")
print(f"Change: ${change:.2f}")
print(f"Whole $5 bills in change: {int(five_bills)}")
print(f"Left after $5 bills: ${leftover:.2f}")
```

Comprehensive Exercise 2: Movie Night

You are planning a movie night for 17 people. Each person gets 3 slices of pizza. Each pizza has 8 slices. The movie is 135 minutes long.

1. Store people, slices per person, slices per pizza, and movie length in variables.
2. Compute total slices needed.
3. Compute how many **whole pizzas** you get with floor division. Also print the leftover slices (`%`). Then print a corrected pizza count: add 1 extra pizza if leftover slices are greater than 0.
4. Convert 135 minutes into hours and minutes using `//` and `%`.
5. Print a summary with f-strings, and print `type()` for at least one `/` result and one `//` result.

In [1]:
```
# Your code here
```

In [1]:

```
# Solution
people = 17
slices_per_person = 3
slices_per_pizza = 8
movie_minutes = 135

total_slices = people * slices_per_person
pizzas_naive = total_slices // slices_per_pizza
remainder_slices = total_slices % slices_per_pizza
pizzas_needed = pizzas_naive
if remainder_slices > 0:
    pizzas_needed = pizzas_naive + 1

hours = movie_minutes // 60
minutes = movie_minutes % 60

print("Movie Night Plan")
print("=" * 28)
print(f"People: {people}")
print(f"Total slices needed: {total_slices}")
print(f"Naive whole pizzas (//): {pizzas_naive}")
print(f"Leftover slices (%): {remainder_slices}")
print(f"Pizzas to order: {pizzas_needed}")
print(f"Movie length: {hours} hour(s) and {minutes}
minute(s)")
print(f"Type of 17 / 8: {type(17 / 8)}")
print(f"Type of 17 // 8: {type(17 // 8)}")
```

Summary

Congratulations! You've completed your first Python programming lecture. Here's what you learned:

Key Concepts

1. **Print Function**: `print()` displays output to the screen
2. **Arithmetic Operators**:
   - `+` Addition
   - `-` Subtraction
   - `*` Multiplication
   - `/` Division (returns float)
   - `//` Floor division (returns integer)
   - `%` Modulus (remainder)
   - `**` Exponentiation (powers)
3. **Variables**: Named containers that store values
4. **Data Types**:
   - `int` - Integers (whole numbers)
   - `float` - Floating-point (decimals)
   - `str` - Strings (text)
   - `bool` - Booleans (True/False)
5. **Strings**: Text data with operations like concatenation, repetition, and methods
6. **F-Strings**: Format strings with `f"text {variable}"` syntax
7. **Math Library**: `import math` for advanced functions like `sqrt()`, `ceil()`, `floor()`

What's Next

In Lecture 2, we'll learn about:

- Making decisions with `if`, `elif`, and `else`
- Comparison operators (`<`, `>`, `==`, etc.)
- Logical operators (`and`, `or`, `not`)
- Getting user input with `input()`

Keep practicing! The best way to learn programming is by writing code.