Introduction to Python Programming

Chapter 3: Input and Print - Introduction

Introduction

Now that you understand variables and data types, it's time to make your programs interactive! This chapter covers how to get information from users (input) and display information to them (output). These are the fundamental building blocks that make programs useful.

---

1. The Print Statement

The `print()` function displays information on the screen. It's one of the most common ways to show output to users.

Basic Print with Strings

Strings can be enclosed in either **single quotes** or **double quotes** - both work the same way:

```
print("Hello, World!")
print('Hello, World!')
```

Why have both?

Having both types of quotes allows you to include one type of quote inside a string:

```
print("She said, 'Hello!'")        # Double quotes outside, single inside
print('He said, "Good morning!"')  # Single quotes outside, double inside
```

Without this flexibility, you'd have problems:

```
print("She said, "Hello!"")  # ERROR! Python gets confused
```

Printing Numbers

You can print numbers directly:

```
print(42)
print(3.14159)
print(100 + 50)  # Prints: 150
```

Printing Variables

```python
name = "Alice"
age = 25
price = 19.99

print(name)    # Alice
print(age)     # 25
print(price)   # 19.99
```

Combining Output: Concatenation vs Commas

There are different ways to combine multiple values in a print statement:

Method 1: Concatenation with +

You can "glue" strings together using the `+` operator:

```python
name = "Bob"
message = "Hello, " + name + "!"
print(message)  # Hello, Bob!

# All in one line
print("Hello, " + name + "!")
```

IMPORTANT LIMITATION: You can only concatenate strings with strings!

```python
age = 25
print("Age: " + age)  # ERROR! Can't concatenate string and integer
```

To fix this, you must convert the number to a string:

```python
age = 25
print("Age: " + str(age))  # Age: 25
```

Method 2: Using Commas

You can separate values with commas in a print statement:

```python
name = "Alice"
age = 25
print("Name:", name, "Age:", age)  # Name: Alice Age: 25
```

Key Difference: Commas automatically:

- Convert values to strings (no need for str())
- Add spaces between items

```python
# With comma - NO casting needed!
age = 25
print("Age:", age)  # Age: 25

# Compare to concatenation - casting required
print("Age: " + str(age))  # Age: 25
```

Method 3: F-Strings (Recommended!)

F-strings (formatted string literals) are the modern, preferred way to format output in Python. They're cleaner, easier to read, and less error-prone.

Add an `f` before the opening quote, then put variables inside `{}`:

```python
name = "Alice"
age = 25
print(f"Name: {name}, Age: {age}")  # Name: Alice, Age: 25
```

Why f-strings are better:

1. **No casting needed** - works with any data type
2. **Much more readable** - no confusing quotes and + signs
3. **Can include expressions** - do calculations right inside

```python
# Example 1: No casting needed
name = "Bob"
age = 30
height = 5.9
print(f"{name} is {age} years old and {height} feet tall")

# Example 2: Expressions inside f-strings
price = 19.99
quantity = 3
print(f"Total: ${price * quantity}")  # Total: $59.97

# Example 3: Calculations
width = 10
height = 5
print(f"Area: {width * height} square feet")  # Area: 50 square feet
```

Comparison of all three methods:

```
name = "Charlie"
age = 28

# Concatenation - requires str(), lots of quotes
print("Hello " + name + ", you are " + str(age) + " years old")

# Commas - automatic conversion, adds spaces
print("Hello", name + ", you are", age, "years old")

# F-string - cleanest and most readable!
print(f"Hello {name}, you are {age} years old")
```

Escape Characters

Special characters in strings that start with a backslash `\` have special meanings:

```
print("Line 1\nLine 2")     # \n = newline
print("Hello\tWorld")       # \t = tab
print("She said \"Hi\"")    # \" = quote inside quotes
print("C:\\Users\\Name")    # \\ = actual backslash
```

We won't use these much right now, but it's good to know they exist. You might see them in code and wonder what they mean. We'll explore these more in later chapters.

---

2. Getting Input from Users

The `input()` function allows users to type information into your program.

Basic Input

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

When this runs:

1. Program waits for user to type something
2. User types their input and presses Enter
3. Whatever they typed gets stored in the `name` variable

CRITICAL CONCEPT: Input Always Returns a String

This is extremely important: No matter what the user types, `input()` ALWAYS gives you a string.

```python
age = input()  # User types: 25
print(type(age))  # <class 'str'> - it's a STRING "25", not the number 25!
```

Why does this matter?

If you want to do math with the input, you MUST convert it:

```python
# WRONG - will cause an error!
age = input()           # User types: 25 (but it's stored as "25")
next_year = age + 1     # ERROR! Can't add string and number

# CORRECT - convert to integer first
age = input()           # User types: 25
age = int(age)          # Convert "25" to 25
next_year = age + 1     # Now this works! 26
```

Input with a Prompt

You can (and should!) include a message to tell users what to enter:

```python
name = input("Enter your name: ")
age = input("Enter your age: ")
```

This is much better than:

```python
print("Enter your name: ")
name = input()
```

Both accomplish the same thing, but putting the prompt inside `input()` is cleaner and more common.

Casting Input

Since `input()` always returns a string, you often need to cast it:

```python
# Get a number as input
age = input("Enter your age: ")
age = int(age)  # Convert string to integer

# Or do it in one line (nested casting)
age = int(input("Enter your age: "))

# For decimal numbers
height = float(input("Enter your height in meters: "))
```

IMPORTANT WARNING: Input Casting Can Crash Your Program!

When you cast user input, you're assuming the user will enter the right type of data. If they don't, your program will crash:

```
age = int(input("Enter your age: "))
# If user types "twenty" instead of "20" → CRASH!
# If user types "20.5" instead of "20" → CRASH!
```

Why does this happen?

```
int("20")      # Works! Returns 20
int("20.5")    # CRASH! Can't convert decimal string to int
int("hello")   # CRASH! Can't convert word to int

float("20.5")  # Works! Returns 20.5
float("hello") # CRASH! Can't convert word to float
```

At this stage of the course, we can't prevent these crashes yet. We don't have the tools (if statements, error handling) to check if the input is valid before converting it.

For now, just be aware:

- Your programs CAN crash if users enter unexpected input
- This is normal at this stage
- Later, we'll learn techniques to validate input and prevent crashes
- When testing your programs, enter the expected type of data

Looking ahead: We can use `type()` to check data types, which will help us validate input once we learn if statements. For now, just know this is a limitation we'll address soon.

---

3. Putting It Together: A Calculator Example

Now that we understand input and output, let's build a simple calculator that demonstrates these concepts:

Basic Two-Number Calculator

```
# Get two numbers from the user
print("=== Simple Calculator ===")
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Perform all basic operations
addition = num1 + num2
subtraction = num1 - num2
multiplication = num1 * num2
```

```
division = num1 / num2
floor_division = num1 // num2
modulus = num1 % num2
exponent = num1 ** num2

# Display results
print(f"\n=== Results ===")
print(f"{num1} + {num2} = {addition}")
print(f"{num1} - {num2} = {subtraction}")
print(f"{num1} * {num2} = {multiplication}")
print(f"{num1} / {num2} = {division}")
print(f"{num1} // {num2} = {floor_division}")
print(f"{num1} % {num2} = {modulus}")
print(f"{num1} ** {num2} = {exponent}")
```

What This Program Does

1. **Gets input:** Asks user for two numbers
2. **Casts input:** Converts strings to floats so we can do math
3. **Performs calculations:** Does all math operations
4. **Displays output:** Shows results using f-strings

Sample Run:

```
=== Simple Calculator ===
Enter first number: 10
Enter second number: 3

=== Results ===
10.0 + 3.0 = 13.0
10.0 - 3.0 = 7.0
10.0 * 3.0 = 30.0
10.0 / 3.0 = 3.3333333333333335
10.0 // 3.0 = 3.0
10.0 % 3.0 = 1.0
10.0 ** 3.0 = 1000.0
```

Another Example: Personal Info Program

```
# Collect user information
name = input("What is your name? ")
age = int(input("What is your age? "))
city = input("What city do you live in? ")

# Calculate some values
```

```python
birth_year = 2024 - age
age_in_months = age * 12

# Display formatted output
print(f"\n=== Your Information ===")
print(f"Name: {name}")
print(f"Age: {age} years old")
print(f"City: {city}")
print(f"You were born around: {birth_year}")
print(f"That's approximately {age_in_months} months!")
```

---

4. Understanding Program Complexity

At this point in the course, we're working with **foundational concepts**. These are the building blocks:

- **Variables** - storing information
- **Data types** - different kinds of information
- **Math operations** - calculating values
- **Input** - getting information from users
- **Output** - displaying information to users

Right now, our programs are fairly simple. They mostly:

- Get some input
- Do some calculations
- Show some output

This is completely normal! These concepts are **easy to understand individually**, but they're incredibly powerful.

The Power of Combination

Here's what makes programming fascinating: When we add **control structures** (which we'll learn soon), simple pieces combine to create complex behavior. It's an **emergent property** - complexity emerges from combining simple parts.

Think of it like LEGO blocks:

- Each block is simple
- But thousands of blocks can build a castle
- The castle is complex, but each block is still simple

In programming:

- `input()` is simple

- `print()` is simple
- Math operations are simple
- But combine them with if statements and loops → suddenly we can build games, calculators, and real programs!

Input and output provide the backbone - they're how programs interact with users. Everything we'll learn next builds on this foundation.

Be patient with yourself! Right now, programs might feel limited. That changes dramatically once we learn:

- **If statements** - making decisions
- **Loops** - repeating actions
- **Functions** - organizing code

Those simple additions will unlock incredible possibilities.

Key Takeaways

✓ **Print displays output** - use f-strings for cleanest formatting

✓ **Input ALWAYS returns a string** - must cast for math operations

✓ **Casting input can crash programs** - if user enters wrong type

✓ **Three ways to format output:** concatenation (+), commas, or f-strings (best!)

✓ **F-strings are recommended:** `f"Hello {name}, you are {age} years old"`

✓ **Include prompts in input():** `input("Enter age: ")` is better than separate print

✓ **These are foundational concepts** - simple now, powerful when combined with control structures

---

Reflection Questions

1. What data type does `input()` always return?
2. If you need to do math with user input, what must you do first?
3. What happens if a user types "hello" when your program expects an integer?
4. What's the difference between using + and , in print statements?
5. Why are f-strings considered the best way to format output?
6. Can you write an input statement that asks for a number and casts it to float in one line?

Practice Exercises

Try building these programs to practice input and output:

1. **Age Calculator:** Ask for birth year, calculate and display current age
2. **Rectangle Calculator:** Get width and height, calculate and display area and perimeter
3. **Temperature Converter:** Get Celsius, convert to Fahrenheit, display both
    - Formula: $F = (C \times 9/5) + 32$
4. **Shopping Total:** Get price and quantity, calculate total with 5% tax
5. **Distance Calculator:** Get speed and time, calculate distance traveled
    - Formula: distance = speed × time

Remember: Your programs might crash if users enter unexpected input - that's okay for now! We'll learn how to handle that soon.