Introduction to Python Programming

Chapter 4: Input and Output - Intermediate

Introduction

This extends Chapter 3 with advanced printing techniques. You'll learn how to control the `print()` function's behavior with `end` and `sep` parameters, master f-string formatting for precise output, and understand raw strings. These tools are essential for creating well-formatted output like multiplication tables, reports, and aligned data.

---

1. The `end` Parameter

By default, `print()` adds a newline character at the end, moving the cursor to the next line. The `end` parameter lets you change this behavior.

Default Behavior

```python
print("Hello")
print("World")
```

Output:

```
Hello
World
```

Each `print()` ends with a newline, so output appears on separate lines.

Changing the End Character

```python
print("Hello", end=" ")
print("World")
```

Output:

```
Hello World
```

Now both prints appear on the same line with a space between them!

Common Uses of `end`

Stay on the same line:

```
print("Loading", end="")
print(".", end="")
print(".", end="")
print(".")
```

Output: `Loading...`

Custom separators:

```
print("Name", end=": ")
print("Alice")
```

Output: `Name: Alice`

No space or newline:

```
for i in range(5):
    print(i, end="")
print()  # Add newline at the end
```

Output: `01234`

The `flush` Parameter

Normally, Python buffers output - it collects characters and sends them in batches for efficiency. The `flush=True` parameter forces immediate output.

Why does this matter?

Without `flush`, you might not see output immediately:

```
import time

# Without flush - might not see dots until the end
print("Loading", end="")
for i in range(3):
    time.sleep(1)
    print(".", end="")
print()
```

With `flush=True`, output appears instantly:

```python
import time

# With flush - see each dot as it appears
print("Loading", end="", flush=True)
for i in range(3):
    time.sleep(1)
    print(".", end="", flush=True)
print()
```

When to use `flush=True`:

- Progress indicators (dots, spinners)
- Real-time status updates
- When using `end=""` with delays
- When you need immediate visual feedback

Practical Example: Progress Indicator

```python
import time

print("Processing", end="", flush=True)
for i in range(5):
    time.sleep(0.5)  # Simulate work
    print(".", end="", flush=True)
print(" Done!")
```

Output: `Processing..... Done!` (each dot appears after 0.5 seconds)

---

2. The `sep` Parameter

The `sep` parameter controls what appears between multiple values in a single `print()` statement.

Default Separator

```python
print("Alice", "Bob", "Charlie")
```

Output: `Alice Bob Charlie`

By default, `sep` is a space: `" "`

Changing the Separator

```python
print("Alice", "Bob", "Charlie", sep=", ")
```

Output: `Alice, Bob, Charlie`

Common Separator Examples

Comma separator:

```
name = "Alice"
age = 25
city = "Calgary"
print(name, age, city, sep=", ")
```

Output: `Alice, 25, Calgary`

Hyphen separator:

```
year = 2024
month = 12
day = 25
print(year, month, day, sep="-")
```

Output: `2024-12-25`

No separator:

```
print("A", "B", "C", sep="")
```

Output: `ABC`

Custom separator:

```
print("Python", "Java", "C++", sep=" | ")
```

Output: `Python | Java | C++`

Combining `sep` and `end`

```
print("Item1", "Item2", "Item3", sep=", ", end=" -> ")
print("Next")
```

Output: `Item1, Item2, Item3 -> Next`

Practical Example: CSV-Style Output

```python
# Create CSV header
print("Name", "Age", "City", sep=",")

# Create CSV rows
print("Alice", 25, "Calgary", sep=",")
print("Bob", 30, "Toronto", sep=",")
print("Charlie", 22, "Vancouver", sep=",")
```

Output:

```
Name,Age,City
Alice,25,Calgary
Bob,30,Toronto
Charlie,22,Vancouver
```

---

3. F-String Formatting Options

F-strings support powerful formatting options for precise control over output appearance.

Basic Format Syntax

Format: `{value:format_spec}`

The format specification comes after a colon inside the braces.

Number Formatting

Decimal places (floating point):

```python
pi = 3.14159265359

print(f"Pi: {pi:.2f}")   # 2 decimal places
print(f"Pi: {pi:.4f}")   # 4 decimal places
print(f"Pi: {pi:.0f}")   # No decimal places (rounds)
```

Output:

```
Pi: 3.14
Pi: 3.1416
Pi: 3
```

Format breakdown:

- `:` starts format specification

- `.2` means 2 digits after decimal
- `f` means fixed-point (floating point) notation

Width and Alignment

Minimum width:

```
num = 42
print(f"Number: {num:5}")   # Right-aligned in 5 spaces
print(f"Number: {num:10}")  # Right-aligned in 10 spaces
```

Output:

```
Number:    42
Number:         42
```

Left alignment:

```
name = "Alice"
print(f"{name:<10}!")  # Left-aligned in 10 spaces
```

Output: `Alice     !`

Right alignment:

```
name = "Alice"
print(f"{name:>10}!")  # Right-aligned in 10 spaces
```

Output: `Alice!`

Center alignment:

```
name = "Alice"
print(f"{name:^10}!")  # Centered in 10 spaces
```

Output: `Alice   !`

Combining Width and Decimal Places

```
price = 19.99
quantity = 3

print(f"Price: ${price:8.2f}")
print(f"Qty:   {quantity:8}")
```

Output:

```
Price: $   19.99
Qty:          3
```

Format breakdown: `{price:8.2f}`

- `8` = minimum total width (including decimal point)
- `.2` = 2 decimal places
- `f` = floating point

Practical Example: Multiplication Table

```python
# Well-formatted multiplication table
print("Multiplication Table (5x5)")
print()

for i in range(1, 6):
    for j in range(1, 6):
        result = i * j
        print(f"{result:4}", end="")
    print()  # New line after each row
```

Output:

```
Multiplication Table (5x5)

   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
```

Each number takes 4 spaces, creating perfect alignment!

Percentage Formatting

```python
score = 0.856

print(f"Score: {score:.1%}")   # 1 decimal place
print(f"Score: {score:.2%}")   # 2 decimal places
```

Output:

```
Score: 85.6%
Score: 85.60%
```

The `%` format multiplies by 100 and adds the % symbol.

Thousands Separator

```python
big_number = 1234567890

print(f"Population: {big_number:,}")
```

Output: `Population: 1,234,567,890`

The `,` adds thousands separators.

Number Systems (Binary, Hex, Octal)

```python
num = 42

print(f"Decimal: {num}")
print(f"Binary:  {num:b}")
print(f"Hex:     {num:x}")
print(f"Octal:   {num:o}")
```

Output:

```
Decimal: 42
Binary:  101010
Hex:     2a
Octal:   52
```

4. Practical Formatting Examples

Example 1: Aligned Table

```python
products = [
    ("Apple", 1.99, 50),
    ("Banana", 0.99, 30),
    ("Orange", 2.49, 40)
]

# Header
print(f"{'Item':<10} {'Price':>8} {'Qty':>6} {'Total':>10}")
print("-" * 36)
```

```
# Rows
for name, price, qty in products:
    total = price * qty
    print(f"{name:<10} ${price:7.2f} {qty:6} ${total:9.2f}")
```

Output:

```
Item          Price    Qty      Total
------------------------------------
Apple        $   1.99      50 $    99.50
Banana       $   0.99      30 $    29.70
Orange       $   2.49      40 $    99.60
```

Example 2: Receipt Generator

```
items = [
    ("Widget", 12.99, 2),
    ("Gadget", 8.50, 1),
    ("Doohickey", 24.99, 1)
]

print("=" * 40)
print(f"{'RECEIPT':^40}")
print("=" * 40)

subtotal = 0
for name, price, qty in items:
    line_total = price * qty
    subtotal += line_total
    print(f"{name:<20} {qty:>3} x ${price:>6.2f} = ${line_total:>7.2f}")

tax = subtotal * 0.05
total = subtotal + tax

print("-" * 40)
print(f"{'Subtotal:':<30} ${subtotal:>7.2f}")
print(f"{'Tax (5%):':<30} ${tax:>7.2f}")
print(f"{'Total:':<30} ${total:>7.2f}")
print("=" * 40)
```

Output:

```
========================================
                RECEIPT
========================================
Widget                 2 x $ 12.99 = $  25.98
Gadget                 1 x $  8.50 = $   8.50
Doohickey              1 x $ 24.99 = $  24.99
----------------------------------------
Subtotal:                       $  59.47
Tax (5%):                       $   2.97
Total:                          $  62.44
========================================
```

Example 3: Progress Bar

```
def print_progress(current, total):
    percentage = (current / total) * 100
    bar_length = 40
    filled = int((current / total) * bar_length)
    bar = "█" * filled + "-" * (bar_length - filled)
    print(f"\r[{bar}] {percentage:5.1f}%", end="", flush=True)

# Simulate progress
import time
for i in range(101):
    print_progress(i, 100)
    time.sleep(0.05)
print()  # New line when done
```

5. Raw Strings (Brief Introduction)

Raw strings treat backslashes as literal characters, not escape sequences. Prefix the string with `r`.

Why Raw Strings?

Regular strings interpret `\` as escape character:

```
path = "C:\new\test"
print(path)  # C:
#ew est (interprets \n and \t as newline and tab!)
```

Problem: `\n` and `\t` are interpreted as escape sequences.

Using Raw Strings

```
path = r"C:\new\test"
print(path)  # C:\new\test (correct!)
```

The `r` prefix tells Python: "treat backslashes literally."

Common Use Cases

File paths (Windows):

```
file_path = r"C:\Users\Alice\Documents\file.txt"
print(file_path)  # C:\Users\Alice\Documents\file.txt
```

Regular expressions (later topic):

```
pattern = r"\d{3}-\d{3}-\d{4}"  # Phone number pattern
```

LaTeX or special formatting:

```
latex = r"\frac{1}{2} \times \pi"
print(latex)  # \frac{1}{2} \times \pi
```

Regular vs Raw Strings

```
# Regular string - backslash is escape
regular = "Line 1\nLine 2\tTabbed"
print(regular)
# Output:
# Line 1
# Line 2    Tabbed

# Raw string - backslash is literal
raw = r"Line 1\nLine 2\tTabbed"
print(raw)
# Output: Line 1\nLine 2\tTabbed
```

When to use raw strings:

- File paths with backslashes
- Regular expressions
- When you need literal backslashes
- When working with escape-heavy text

When NOT to use raw strings:

- When you want escape sequences (`\n`, `\t`) to work
- Most normal string operations

6. Combining Everything

Example: Formatted Multiplication Table

```python
# Enhanced multiplication table with headers
size = 10

# Print header row
print("  ", end="")
for i in range(1, size + 1):
    print(f"{i:4}", end="")
print()

# Print separator
print("  " + "-" * (size * 4))

# Print table rows
for i in range(1, size + 1):
    print(f"{i:2}|", end="")
    for j in range(1, size + 1):
        print(f"{i*j:4}", end="")
    print()
```

Output:

```
      1   2   3   4   5   6   7   8   9  10
  ----------------------------------------
 1|   1   2   3   4   5   6   7   8   9  10
 2|   2   4   6   8  10  12  14  16  18  20
 3|   3   6   9  12  15  18  21  24  27  30
...
```

Key Takeaways

✓ `end` **parameter** controls what prints after output (default is newline)

✓ `sep` **parameter** controls what appears between values (default is space)

✓ **F-string formatting:** `{value:width.precision}`

✓ **Decimal places:** `{num:.2f}` for 2 decimal places

✓ **Alignment:** `<` left, `>` right, `^` center

✓ **Width:** `{value:10}` reserves 10 spaces

✓ **Thousands separator:** `{num:,}` adds commas

✓ **Percentage:** `{num:.1%}` formats as percentage

✓ **Raw strings:** `r"text"` treats backslashes literally

✓ **Combine techniques** for professional-looking output

Reflection Questions

1. What does `print("Hi", end="")` do differently from `print("Hi")`?
2. How do you make `print()` separate values with commas instead of spaces?
3. What does `{value:.3f}` mean in an f-string?
4. How do you right-align a number in 10 spaces?
5. When should you use raw strings?
6. How do you format a number with 2 decimal places and thousands separators?
7. What's the difference between `{num:5}` and `{num:.5f}`?
8. How do you keep multiple prints on the same line?

Practice Exercises

1. **Number Formatter:** Print a number with 0, 2, and 4 decimal places
2. **Alignment Practice:** Print three strings left, center, and right aligned in 20 spaces each
3. **Price List:** Create a formatted price list with items, prices (2 decimals), and quantities aligned
4. **Countdown:** Print numbers 10 to 1 on the same line separated by spaces
5. **Percentage Table:** Show numbers 0.1 to 0.9 as percentages with 1 decimal place
6. **File Path:** Use a raw string to print a Windows file path correctly
7. **Multiplication Table:** Create a formatted 12×12 multiplication table with aligned columns
8. **Progress Bar:** Create a simple text-based progress bar using `end` parameter
9. **CSV Export:** Format student data as comma-separated values
10. **Invoice:** Create a formatted invoice with aligned columns for items, quantities, prices, and totals