Tutorial by Dr. Ayes Chinmay
Date: 25/09/2025

Chapter 5: Formatting Text Precisely - Comprehensive Tutorial

Introduction

Text formatting is a fundamental skill in Python programming that allows developers to display data in a user-friendly and organized manner. Chapter 5 covers various methods to format text precisely, including the percent sign operator, the format function, and the format method. This tutorial provides comprehensive coverage of all sections (5.1 through 5.9) with practical code examples and their corresponding outputs.

5.1 Formatting with the Percent Sign Operator (%)

The percent sign operator is one of the oldest and most straightforward methods for string formatting in Python. It works by placing format specifiers within a string and then providing values to fill those placeholders.

Basic Concept

The percent operator uses a tuple on the right side to provide values for substitution. The general syntax is:

```
"format_string" % (values)
```

Code Example 5.1.1: Basic Percent Sign Formatting

```
# Basic string formatting with %s
name = "Alice"
age = 28
height = 5.7

result = "Name: %s, Age: %d, Height: %.1f" % (name, age, height)
print(result)
```

Output:

```
Name: Alice, Age: 28, Height: 5.7
```

Code Example 5.1.2: Multiple Substitutions

```
# Multiple substitutions using percent operator
city = "New York"
population = 8335897
temperature = 72.5

formatted_string = "The city %s has a population of %d and temperature of %.2f°F" % (city
print(formatted_string)
```

Output:

```
The city New York has a population of 8335897 and temperature of 72.50°F
```

Code Example 5.1.3: Repeated Values

```
# Using the same value multiple times
value = "Python"
result = "I love %s. %s is awesome. %s is powerful." % (value, value, value)
print(result)
```

Output:

```
I love Python. Python is awesome. Python is powerful.
```

5.2 Percent Sign (%) Format Specifiers

Format specifiers define how values should be displayed. Each specifier is preceded by the percent sign (%) and followed by a letter indicating the type.

Understanding Format Specifiers

Specifier	Meaning	Example
%s	String	"Hello"
%d	Integer	42
%f	Float	3.14159
%e	Scientific notation	3.14e+00
%x	Hexadecimal	2a
%o	Octal	52
%%	Percent sign	%

Code Example 5.2.1: String Formatting with %s

```
# String formatting
first_name = "John"
last_name = "Smith"
occupation = "Engineer"

output = "%s %s is an %s" % (first_name, last_name, occupation)
print(output)
```

Output:

```
John Smith is an Engineer
```

Code Example 5.2.2: Integer Formatting with %d

```
# Integer formatting
quantity = 150
price_per_unit = 25
total_price = 3750

statement = "Quantity: %d, Unit Price: %d, Total: %d" % (quantity, price_per_unit, total_
print(statement)
```

Output:

```
Quantity: 150, Unit Price: 25, Total: 3750
```

Code Example 5.2.3: Float Formatting with %f

```
# Float formatting with default precision
value1 = 3.14159
value2 = 2.71828
value3 = 1.41421

output = "Values: %f, %f, %f" % (value1, value2, value3)
print(output)
```

Output:

```
Values: 3.141590, 2.718280, 1.414210
```

Code Example 5.2.4: Hexadecimal, Octal, and Scientific Notation

```
# Various number formats
decimal_num = 255

hex_format = "Hexadecimal: %x" % decimal_num
octal_format = "Octal: %o" % decimal_num
sci_format = "Scientific: %e" % decimal_num

print(hex_format)
print(octal_format)
print(sci_format)
```

Output:

```
Hexadecimal: ff
Octal: 377
Scientific: 2.550000e+02
```

Code Example 5.2.5: Percentage Sign in Output

```
# Including percent sign in output
score = 85
max_score = 100
percentage = 85.0

output = "Your score is %d out of %d, which is %.1f%%" % (score, max_score, percentage)
print(output)
```

Output:

```
Your score is 85 out of 100, which is 85.0%
```

5.3 Percent Sign Variable-Length Print Fields

Variable-length print fields allow you to control the width of the output field. This is useful for creating aligned columns or maintaining consistent spacing.

Field Width Specification

The width specifier is placed between the percent sign and the format type. By default, values are right-aligned. Use a negative width for left-alignment.

Code Example 5.3.1: Right-Aligned Fields

```
# Right-aligned fields with width specification
names = ["Alice", "Bob", "Charlie"]
scores = [95, 87, 92]

print("Name         Score")
print("-" * 20)
for name, score in zip(names, scores):
    print("%15s %5d" % (name, score))
```

Output:

```
Name         Score
--------------------
          Alice    95
            Bob    87
        Charlie    92
```

Code Example 5.3.2: Left-Aligned Fields

```
# Left-aligned fields using negative width
items = ["Apples", "Bananas", "Oranges"]
prices = [1.25, 0.99, 1.50]

print("Item          Price")
print("-" * 25)
for item, price in zip(items, prices):
    print("%-15s $%.2f" % (item, price))
```

Output:

```
Item          Price
-------------------------
Apples          $1.25
Bananas         $0.99
Oranges         $1.50
```

Code Example 5.3.3: Variable Width with Floating-Point Numbers

```
# Width and precision for floating-point numbers
values = [3.14159, 2.71828, 1.41421]

print("Standard Width:")
for value in values:
    print("%10.2f" % value)

print("\nSmaller Width:")
```

```
for value in values:
    print("%8.3f" % value)
```

Output:

```
Standard Width:
      3.14
      2.72
      1.41

Smaller Width:
   3.142
   2.718
   1.414
```

Code Example 5.3.4: Zero-Padded Numbers

```
# Zero-padding for numeric fields
account_numbers = [12345, 67890, 54321]
amounts = [1000.50, 2500.75, 3200.25]

print("Account Number    Amount")
print("-" * 30)
for acc_num, amount in zip(account_numbers, amounts):
    print("%010d        %8.2f" % (acc_num, amount))
```

Output:

```
Account Number    Amount
------------------------------
0000012345        1000.50
0000067890        2500.75
0000054321        3200.25
```

5.4 The Global Format Function

The `format()` function is a built-in Python function that provides a more flexible and modern approach to string formatting compared to the percent operator.

Basic Syntax

```
format(value, format_spec)
```

Code Example 5.4.1: Basic Format Function

```
# Using the format() function
name = "David"
age = 35

formatted = format(name, '') + " is " + format(age, 'd') + " years old"
print(formatted)

# More practical usage
print("{} is {} years old".format(name, age))
```

Output:

```
David is 35 years old
David is 35 years old
```

Code Example 5.4.2: Format Specifications

```
# Format function with specifications
value = 3.14159

print("Default: {}".format(value))
print("2 decimal places: {:.2f}".format(value))
print("Scientific notation: {:.2e}".format(value))
print("Percentage: {:.1%}".format(value))
```

Output:

```
Default: 3.14159
2 decimal places: 3.14
Scientific notation: 3.14e+00
Percentage: 314.2%
```

Code Example 5.4.3: Positional Arguments

```
# Using positional arguments with format()
template = "{} {} {} {} {}"
result = template.format("One", "Two", "Three", "Four", "Five")
print(result)

# Accessing specific positions
print("{3} {1} {4} {0} {2}".format("A", "B", "C", "D", "E"))
```

Output:

```
One Two Three Four Five
```

```
D B E A C
```

5.5 Introduction to the Format Method

The format method is a string method that works similarly to the global `format()` function but is called directly on the string object.

Basic Syntax

```
"template_string".format(args)
```

Code Example 5.5.1: Basic Format Method

```
# Using the format method
product = "Laptop"
price = 999.99
quantity = 2

message = "Product: {}, Price: ${}, Quantity: {}".format(product, price, quantity)
print(message)
```

Output:

```
Product: Laptop, Price: $999.99, Quantity: 2
```

Code Example 5.5.2: Format Method with Type Specifiers

```
# Using type specifiers with format method
integer_val = 42
float_val = 3.14159
string_val = "Python"

output = "Int: {:d}, Float: {:.2f}, String: {}".format(integer_val, float_val, string_val
print(output)
```

Output:

```
Int: 42, Float: 3.14, String: Python
```

Code Example 5.5.3: Multiple Format Specifications

```
# Complex formatting with format method
data = [
    ("Alice", 95, 4.0),
    ("Bob", 87, 3.8),
    ("Charlie", 92, 3.9)
]

print("Name          Grade    GPA")
print("-" * 30)
for name, grade, gpa in data:
    print("{:&lt;15} {:&gt;5}  {:&gt;4.2f}".format(name, grade, gpa))
```

Output:

```
Name          Grade    GPA
------------------------------
Alice            95   4.00
Bob              87   3.80
Charlie          92   3.90
```

5.6 Ordering by Position, Name, or Number

The format method and function allow you to specify values by position, assign names using keyword arguments, or reference the same value multiple times.

Code Example 5.6.1: Positional Ordering

```
# Positional ordering in format method
colors = ["red", "green", "blue"]

print("Using order 0, 1, 2:")
print("{} {} {}".format(colors[^0], colors[^1], colors[^2]))

print("Using order 2, 1, 0:")
print("{2} {1} {0}".format(colors[^0], colors[^1], colors[^2]))

print("Repeating positions:")
print("{0} {1} {0} {2}".format(colors[^0], colors[^1], colors[^2]))
```

Output:

```
Using order 0, 1, 2:
red green blue
Using order 2, 1, 0:
blue green red
Repeating positions:
red green red blue
```

Code Example 5.6.2: Named Arguments

```
# Using named arguments with format method
person_info = {
    "name": "Sarah",
    "age": 30,
    "city": "Boston",
    "occupation": "Doctor"
}

template = "{name} is {age} years old and works as a {occupation} in {city}."
result = template.format(**person_info)
print(result)

# Direct named arguments
print("{first} {last} is from {place}".format(first="John", last="Doe", place="USA"))
```

Output:

```
Sarah is 30 years old and works as a Doctor in Boston.
John Doe is from USA
```

Code Example 5.6.3: Combining Positional and Named Arguments

```
# Mixing positional and named arguments
template = "{0} likes {color} and {1} likes {fruit}"
result = template.format("Alice", "Bob", color="blue", fruit="apple")
print(result)
```

Output:

```
Alice likes blue and Bob likes apple
```

5.7 repr() Versus String Conversion

Understanding the difference between `repr()` and string conversion (`str()`) is crucial for proper formatting and debugging.

Key Differences

- `str()` returns a user-friendly string representation
- `repr()` returns a developer-friendly string representation that could theoretically be used to recreate the object

Code Example 5.7.1: String Conversion vs repr()

```
# Demonstrating str() vs repr()
text = "Hello World"
number = 42

print("Using str():")
print(str(text))
print(str(number))

print("\nUsing repr():")
print(repr(text))
print(repr(number))
```

Output:

```
Using str():
Hello World
42

Using repr():
'Hello World'
42
```

Code Example 5.7.2: Multiline Strings and repr()

```
# Multiline strings with str() and repr()
multiline = "Line 1\nLine 2\nLine 3"

print("Using str():")
print(str(multiline))

print("\nUsing repr():")
print(repr(multiline))
```

Output:

```
Using str():
Line 1
Line 2
Line 3

Using repr():
'Line 1\nLine 2\nLine 3'
```

Code Example 5.7.3: Special Characters with repr()

```
# Special characters representation
text_with_quotes = 'She said "Hello"'
text_with_tab = "Column1\tColumn2"

print("String conversion:")
print(str(text_with_quotes))
print(str(text_with_tab))

print("\nRepr conversion:")
print(repr(text_with_quotes))
print(repr(text_with_tab))
```

Output:

```
String conversion:
She said "Hello"
Column1 Column2

Repr conversion:
'She said "Hello"'
'Column1\tColumn2'
```

5.8 The Spec Field of the Format Function and Method

The format specification mini-language provides fine-grained control over how values are formatted. The general format of a format spec is:

```
[[fill]align][sign][#][^0][width][,][.precision][type]
```

5.8.1 Print-Field Width

The width field specifies the minimum width of the output field.

Code Example 5.8.1: Field Width

```
# Controlling field width
values = [42, 3.14159, "Python"]

print("Width 10:")
print("{:10}".format(values[^0]))
print("{:10}".format(values[^1]))
print("{:10}".format(values[^2]))

print("\nWidth 15:")
print("{:15}".format(values[^0]))
```

```
print("{:15}".format(values[^1]))
print("{:15}".format(values[^2]))
```

Output:

```
Width 10:
        42
    3.14159
    Python

Width 15:
             42
          3.14159
          Python
```

5.8.2 Text Justification: fill and align Characters

The alignment options are: < (left), > (right), ^ (center), and = (for numbers with sign/prefix).

Code Example 5.8.2: Text Alignment

```
# Text alignment and fill characters
text = "Python"

print("Left align (default fill with space):")
print("|{:&lt;10}|".format(text))

print("\nRight align (default fill with space):")
print("|{:&gt;10}|".format(text))

print("\nCenter align:")
print("|{:^12}|".format(text))

print("\nLeft align with dot fill:")
print("|{:.&lt;10}|".format(text))

print("\nCenter align with asterisk fill:")
print("|{:*^12}|".format(text))
```

Output:

```
Left align (default fill with space):
|Python    |

Right align (default fill with space):
|    Python|

Center align:
|   Python   |

Left align with dot fill:
```

```
|Python....|

Center align with asterisk fill:
|***Python***|
```

5.8.3 The sign Character

The sign option controls the display of the sign for numeric values: + (show sign for all), - (show only negative), or space (space for positive).

Code Example 5.8.3: Sign Control

```
# Sign control for numbers
pos_num = 42
neg_num = -42

print("Show sign for all (+):")
print("{:+d}".format(pos_num))
print("{:+d}".format(neg_num))

print("\nShow sign only for negative (-):")
print("{:-d}".format(pos_num))
print("{:-d}".format(neg_num))

print("\nSpace for positive:")
print("{: d}".format(pos_num))
print("{: d}".format(neg_num))
```

Output:

```
Show sign for all (+):
+42
-42

Show sign only for negative (-):
42
-42

Space for positive:
 42
-42
```

5.8.4 The Leading-Zero Character (0)

Zero-padding is useful for numeric values and fixed-width fields.

Code Example 5.8.4: Zero-Padding

```
# Zero-padding for numbers
numbers = [42, 123, 7]

print("Zero-padded to width 5:")
for num in numbers:
    print("{:05d}".format(num))

print("\nZero-padded floats:")
print("{:08.2f}".format(3.14))
print("{:08.2f}".format(25.7))
```

Output:

```
Zero-padded to width 5:
00042
00123
00007

Zero-padded floats:
00003.14
00025.70
```

5.8.5 Thousands Place Separator

The comma (,) separator formats large numbers with commas for readability.

Code Example 5.8.5: Thousands Separator

```
# Using comma as thousands separator
large_numbers = [1000, 1000000, 123456789]

print("Without separator:")
for num in large_numbers:
    print("{}".format(num))

print("\nWith comma separator:")
for num in large_numbers:
    print("{:,}".format(num))

print("\nWith separator and zero-padding:")
print("{:010,}".format(1000))
print("{:015,}".format(1000000))
```

Output:

```
Without separator:
1000
1000000
123456789
```

```
With comma separator:
1,000
1,000,000
123,456,789

With separator and zero-padding:
0001,000
00001,000,000
```

5.8.6 Controlling Precision

Precision specifies the number of digits after the decimal point for floating-point numbers.

Code Example 5.8.6: Precision Control

```
# Controlling precision for floats
pi = 3.14159265359

print("Default precision:")
print("{}".format(pi))

print("\n1 decimal place:")
print("{:.1f}".format(pi))

print("\n3 decimal places:")
print("{:.3f}".format(pi))

print("\n6 decimal places:")
print("{:.6f}".format(pi))
```

Output:

```
Default precision:
3.14159265359

1 decimal place:
3.1

3 decimal places:
3.142

6 decimal places:
3.141593
```

5.8.7 Precision Used with Strings (Truncation)

Precision can also truncate strings to a maximum length.

Code Example 5.8.7: String Truncation

```
# Using precision to truncate strings
text = "Python Programming Language"

print("Full string:")
print("{}".format(text))

print("\nTruncated to 10 characters:")
print("{:.10}".format(text))

print("\nTruncated to 15 characters with left alignment:")
print("{:&amp;lt;.15}".format(text))

print("\nTruncated to 12 characters with center alignment and dot fill:")
print("{:^.12}".format(text))
```

Output:

```
Full string:
Python Programming Language

Truncated to 10 characters:
Python Prog

Truncated to 15 characters with left alignment:
Python Programm

Truncated to 12 characters with center alignment and dot fill:
..Python Prog..
```

5.8.8 Type Specifiers

Type specifiers determine how the value is interpreted and displayed.

Code Example 5.8.8: Type Specifiers

```
# Various type specifiers
num = 42
float_num = 3.14

print("Integer types:")
print("Decimal (d): {:d}".format(num))
print("Octal (o): {:o}".format(num))
print("Hex lowercase (x): {:x}".format(num))
print("Hex uppercase (X): {:X}".format(num))
```

```
print("\nFloating-point types:")
print("Fixed-point (f): {:.2f}".format(float_num))
print("Exponent (e): {:.2e}".format(float_num))
print("Exponent uppercase (E): {:.2E}".format(float_num))
print("General format (g): {:.2g}".format(float_num))
```

Output:

```
Integer types:
Decimal (d): 42
Octal (o): 52
Hex lowercase (x): 2a
Hex uppercase (X): 2A

Floating-point types:
Fixed-point (f): 3.14
Exponent (e): 3.14e+00
Exponent uppercase (E): 3.14E+00
General format (g): 3.1
```

5.8.9 Displaying in Binary Radix

The binary format specifier (b) displays integers in binary.

Code Example 5.8.9: Binary Display

```
# Binary representation
numbers = [10, 255, 1024]

print("Binary representation:")
for num in numbers:
    print("{} in binary: {:b}".format(num, num))

print("\nBinary with zero-padding:")
print("{:08b}".format(10))
print("{:08b}".format(15))
print("{:08b}".format(255))
```

Output:

```
Binary representation:
10 in binary: 1010
255 in binary: 11111111
1024 in binary: 10000000000

Binary with zero-padding:
00001010
00001111
11111111
```

5.8.10 Displaying in Octal and Hex Radix

Octal (o) and hexadecimal (x, X) representations are commonly used in programming.

Code Example 5.8.10: Octal and Hex Display

```
# Octal and hexadecimal representations
num = 255

print("Number: {}".format(num))
print("Octal: {:o}".format(num))
print("Octal with prefix: {:#o}".format(num))
print("Hex (lowercase): {:x}".format(num))
print("Hex (uppercase): {:X}".format(num))
print("Hex with prefix: {:#x}".format(num))

print("\nTable of conversions:")
print("Dec  Oct   Hex")
print("-" * 15)
for i in [10, 50, 100, 255, 256]:
    print("{:3d} {:4o}  {:3x}".format(i, i, i))
```

Output:

```
Number: 255
Octal: 377
Octal with prefix: 0o377
Hex (lowercase): ff
Hex (uppercase): FF
Hex with prefix: 0xff

Table of conversions:
Dec  Oct   Hex
---------------
 10   12    a
 50   62   32
100  144   64
255  377   ff
256  400  100
```

5.8.11 Displaying Percentages

The percentage format (%) displays numbers as percentages.

Code Example 5.8.11: Percentage Display

```
# Percentage formatting
values = [0.75, 0.333, 0.95, 1.5, 0.05]

print("As percentages:")
for value in values:
```

```
    print("{:.1%}".format(value))

print("\nPercentages with no decimal places:")
for value in values:
    print("{:.0%}".format(value))

print("\nTest scores as percentages:")
scores = [85/100, 92/100, 78/100, 95/100]
for i, score in enumerate(scores, 1):
    print("Student {}: {:.1%}".format(i, score))
```

Output:

```
As percentages:
75.0%
33.3%
95.0%
150.0%
5.0%

Percentages with no decimal places:
75%
33%
95%
150%
5%

Test scores as percentages:
Student 1: 85.0%
Student 2: 92.0%
Student 3: 78.0%
Student 4: 95.0%
```

5.8.12 Binary Radix Example

A comprehensive example combining binary display with formatting.

Code Example 5.8.12: Comprehensive Binary Example

```
# Comprehensive binary formatting example
print("Number | Binary (8-bit) | Binary (16-bit) | Hex   | Octal")
print("-" * 60)

numbers = [0, 1, 15, 16, 127, 128, 255, 256]

for num in numbers:
    binary_8 = "{:08b}".format(num % 256)  # Keep within 8 bits
    binary_16 = "{:016b}".format(num)
    hex_val = "{:04x}".format(num)
    octal_val = "{:04o}".format(num)
    print("{:6d} | {:14s} | {:15s} | {:5s} | {:5s}".format(
        num, binary_8, binary_16, hex_val, octal_val))
```

```
print("\nBitwise operations representation:")
a = 12  # 1100
b = 10  # 1010
print("a = {:04b} ({})".format(a, a))
print("b = {:04b} ({})".format(b, b))
print("a &amp;amp; b = {:04b} ({})".format(a &amp;amp; b, a &amp;amp; b))
print("a | b = {:04b} ({})".format(a | b, a | b))
print("a ^ b = {:04b} ({})".format(a ^ b, a ^ b))
```

Output:

```
Number | Binary (8-bit) | Binary (16-bit) | Hex   | Octal
------------------------------------------------------------
     0 | 00000000       | 0000000000000000 | 0000  | 0000
     1 | 00000001       | 0000000000000001 | 0001  | 0001
    15 | 00001111       | 0000000000001111 | 000f  | 0017
    16 | 00010000       | 0000000000010000 | 0010  | 0020
   127 | 01111111       | 0000000001111111 | 007f  | 0177
   128 | 10000000       | 0000000010000000 | 0080  | 0200
   255 | 11111111       | 0000000011111111 | 00ff  | 0377
   256 | 00000000       | 0000000100000000 | 0100  | 0400

Bitwise operations representation:
a = 1100 (12)
b = 1010 (10)
a &amp;amp; b = 1000 (8)
a | b = 1110 (14)
a ^ b = 0110 (6)
```

5.9 Variable-Size Fields

Variable-size fields use dynamic width and precision, allowing formatting specifications to be derived from the data itself or from separate arguments.

Code Example 5.9.1: Dynamic Width

```
# Dynamic width specification using *
values = ["Hello", "World", "Python"]
widths = [10, 15, 20]

print("Dynamic width formatting:")
for value, width in zip(values, widths):
    print("|{:*^{}}|".format(value, width))

print("\nUsing * for width:")
print("{:*^{}}".format("Center", 20))
print("{:*&amp;gt;{}}".format("Right", 15))
print("{:*&amp;lt;{}}".format("Left", 15))
```

Output:

```
Dynamic width formatting:
|*****Hello*****|
|*****World*****|
|*****Python*****|

Using * for width:
*****Center*****
**********Right
Left**********
```

Code Example 5.9.2: Dynamic Precision

```
# Dynamic precision specification
pi_value = 3.14159265359
e_value = 2.71828182846

print("Dynamic precision formatting:")
precisions = [1, 3, 5]

for precision in precisions:
    result = "{:.{}}".format(pi_value, precision)
    print("Precision {}: {}".format(precision, result))

print("\nFormatting multiple values:")
values = [3.14159, 2.71828, 1.41421]
for precision in [1, 2, 3]:
    formatted = " ".join("{:.{}}".format(v, precision) for v in values)
    print("Precision {}: {}".format(precision, formatted))
```

Output:

```
Dynamic precision formatting:
Precision 1: 3.1
Precision 3: 3.142
Precision 5: 3.14159

Formatting multiple values:
Precision 1: 3.1 2.7 1.4
Precision 2: 3.14 2.72 1.41
Precision 3: 3.142 2.718 1.414
```

Code Example 5.9.3: Combining Dynamic Width and Precision

```
# Combining dynamic width and precision
data_points = [3.14159, 25.7, 1000.5]
widths = [10, 12, 15]
precisions = [2, 1, 3]

print("Table with dynamic width and precision:")
print("Value        Width   Precision   Formatted")
```

```
print("-" * 50)

for value, width, precision in zip(data_points, widths, precisions):
    formatted = "{:*.{}}".format(value, precision, width)
    print("{:&lt;12} {:&gt;5}   {:&gt;9}     {}".format(value, width, precision, formatte
```

Output:

```
Table with dynamic width and precision:
Value        Width   Precision   Formatted
--------------------------------------------------
3.14159        10          2        3.14
25.7           12          1        25.7
1000.5         15          3        1000.500
```

Code Example 5.9.4: Using Named Arguments for Dynamic Fields

```
# Dynamic fields using named arguments
template = "{value:*^{width}}"

print("Dynamic fields with named arguments:")
print(template.format(value="Python", width=20))
print(template.format(value="Tutorial", width=25))
print(template.format(value="Guide", width=15))

print("\nDynamic precision with named arguments:")
template2 = "{value:.{precision}f}"
print(template2.format(value=3.14159, precision=2))
print(template2.format(value=3.14159, precision=4))
print(template2.format(value=3.14159, precision=6))
```

Output:

```
Dynamic fields with named arguments:
*****Python*****
*******Tutorial*******
*****Guide*****

Dynamic precision with named arguments:
3.14
3.1416
3.141593
```

Code Example 5.9.5: Advanced Variable-Size Field Example

```
# Advanced variable-size field formatting
headers = ["Name", "Score", "Grade"]
data = [
    ("Alice", 95, "A"),
    ("Bob", 87, "B"),
```

```
    ("Charlie", 92, "A"),
    ("Diana", 78, "C")
]

# Calculate column widths
col_widths = [max(len(header), max(len(str(row[i])) for row in data))
              for i, header in enumerate(headers)]

# Print header
header_format = " | ".join("{{:^{}}}".format(width) for width in col_widths)
print(header_format.format(*headers))
print("-" * (sum(col_widths) + len(col_widths) * 3 - 1))

# Print data rows
row_format = " | ".join("{{:^{}}}".format(width) for width in col_widths)
for row in data:
    print(row_format.format(*row))
```

Output:

```
 Name  | Score | Grade
-------+-------+-------
Alice  |  95   |   A
 Bob   |  87   |   B
Charlie|  92   |   A
Diana  |  78   |   C
```

Chapter Summary

Chapter 5 provided comprehensive coverage of text formatting in Python, from the legacy percent sign operator to modern format methods and functions. The key topics covered include:

1. **Percent Sign Operator (%)**: The oldest method of string formatting using format specifiers like %s, %d, and %f.
2. **Format Specifiers**: Various specifiers for controlling data types and basic formatting options.
3. **Variable-Length Print Fields**: Controlling field width for aligned columns and consistent spacing.
4. **Global Format Function**: The flexible `format()` function for converting values with specifications.
5. **Format Method**: String method for formatting providing similar functionality to the global format function.
6. **Positional and Named Arguments**: Ordering values by position or using keyword arguments for clarity.
7. **repr() vs str()**: Understanding the difference between developer-friendly and user-friendly representations.
8. **Format Specification Mini-Language**: Comprehensive control over alignment, signs, padding, precision, and type specifiers.

9. **Advanced Number Formats**: Binary, octal, and hexadecimal representations, along with percentage formatting.
10. **Variable-Size Fields**: Dynamic width and precision based on runtime values.

Mastering these formatting techniques is essential for producing well-organized, readable output in Python programs.

Review Questions

1. What is the difference between the percent sign operator and the format method?
2. How do you use the format method to display a number with two decimal places?
3. What does the `{:>10}` format specifier do?
4. How can you display a number in hexadecimal format?
5. What is the difference between `str()` and `repr()`?
6. How do you create a variable-width format field?
7. What does the comma separator (,) do in format specifications?
8. How do you display a number as a percentage?
9. Explain the difference between `{:05d}` and `{:5d}`.
10. How do you truncate a string to 10 characters using format?

Suggested Practice Problems

1. Create a program that formats student records with aligned columns for name, ID, and GPA.
2. Write a function that converts numbers to different bases (binary, octal, hexadecimal) and displays them with proper formatting.
3. Build a program that generates a formatted multiplication table with proper alignment.
4. Create a financial report formatter that displays currency values with proper formatting and thousand separators.
5. Write a program that demonstrates all format specifiers with clear examples and explanations.
6. Create a table formatter that dynamically adjusts column widths based on data content.
7. Build a number formatting utility that supports multiple precision levels and thousands separators.
8. Write a program that compares the output of percent formatting and format method on various data types.
9. Create a string truncation tool with different alignment options.
10. Build a custom formatter class that extends Python's formatting capabilities with additional features.

⁂