Universidad ICESI

String Manipulation and Formatting

Computation and Discrete Structures III

Juan Marcos Caicedo Mejía – jmcaicedo@icesi.edu.co

Department of Computing and Intelligent Systems
Barberi Faculty of Engineering, Design and Applied Sciences
ICESI University

Material adapted from DataCamp - Regular expressions in Python

2026

Contents

Contenido

You will learn

- String manipulation
- String formatting
- Basic and advanced regular expressions

Strings are sequences of characters. In this section, we explore how to modify, inspect, and transform them using built-in Python methods.

Why it is important

- Clean dataset for text mining or sentiment analysis
- Process email content to detect spam
- Parse and extract data from websites

Text data is everywhere. Being able to manipulate strings efficiently is essential for data cleaning and preprocessing.

Strings

A string is a sequence of characters enclosed in single or double quotes.

```
my_string = "This is a string"
my_string2 = 'This is also a string'
my_string = "And this? It's the correct string"
```

Both single and double quotes work, but they must match.

Length and Conversion

The function `len()` returns the number of characters in a string. The function `str()` converts other data types into strings.

```
my_string = "Awesome day"
len(my_string)

str(123)
```

Concatenation

Concatenation means joining strings together using the + operator.

```
my_string1 = "Awesome day"
my_string2 = "for biking"
print(my_string1 + " " + my_string2)
```

Strings are immutable, so concatenation creates a new string.

Indexing

Indexing allows access to individual characters using bracket notation.

```
my_string = "Awesome day"

print(my_string[3])
print(my_string[-1])
```

Indexing starts at 0. Negative indices count from the end.

Slicing

Slicing extracts a substring using the syntax `[start:end]`.

```
my_string = "Awesome day"

print(my_string[0:3])
print(my_string[:5])
print(my_string[5:])
```

The start index is inclusive, the end index is exclusive.

Stride

Stride allows skipping characters using `[start:end:step]`.

```
my_string = "Awesome day"

print(my_string[0:6:2])
print(my_string[::-1])
```

A negative step reverses the string.

String Operations

Python provides many built-in methods to manipulate strings.

- Change letter cases
- Split and join text
- Remove unwanted spaces

Adjusting Cases

These methods modify letter casing.

```
my_string = "tHis Is a niCe StriNg"

print(my_string.lower())
print(my_string.upper())
print(my_string.capitalize())
```

They return new strings without modifying the original.

Splitting

`split()` breaks a string into a list of substrings.

```
my_string = "This string will be split"

my_string.split(sep=" ", maxsplit=2)
my_string.rsplit(sep=" ", maxsplit=2)
```

You can control the separator and the maximum number of splits.

Split Lines

`splitlines()` breaks a string at line boundaries.

```
my_string = "This string will be split\nin two"

my_string.splitlines()
```

Useful when processing text files.

Joining

`join()` combines elements of an iterable into a single string.

```
my_list = ["this", "would", "be", "a", "string"]

print(" ".join(my_list))
```

The string before `.join()` is used as a separator.

Stripping Characters

Stripping removes whitespace or specified characters from the ends.

```
my_string = " This string will be stripped\n"

my_string.strip()
my_string.rstrip()
my_string.lstrip()
```

`strip()` removes both sides, `lstrip()` left, `rstrip()` right.

Finding and Replacing

These methods help search, count, and modify substrings inside a string.

Finding Substrings

`find()` returns the index of the first occurrence or -1 if not found.

```
my_string = "Where's Waldo?"

my_string.find("Waldo")
my_string.find("Wenda")
```

Finding with Range

You can restrict the search to a specific slice of the string.

```
my_string = "Where's Waldo?"

my_string.find("Waldo", 0, 6)
```

Index Function

`index()` works like `find()`, but raises an error if not found.

```
my_string = "Where's Waldo?"

my_string.index("Waldo")
my_string.index("Wenda")
```

Index with Try/Except

Exception handling prevents program crashes.

```
my_string = "Where's Waldo?"

try:
    my_string.index("Wenda")
except ValueError:
    print("Not found")
```

Counting Occurrences

`count()` returns how many times a substring appears.

```
my_string = "How many fruits do you have in your fruit basket?"

my_string.count("fruit")
my_string.count("fruit", 0, 16)
```

Replacing Substrings

`replace()` substitutes occurrences of a substring.

```
my_string = "The red house is between the blue house and the old house"

print(my_string.replace("house", "car"))
print(my_string.replace("house", "car", 2))
```

The optional third argument limits the number of replacements.

Wrapping Up

- Slice and concatenate strings
- Modify letter cases
- Split and join text
- Remove unwanted characters
- Search and replace substrings

Strings are immutable, but Python provides powerful methods to transform them.

Contenido

What is String Formatting?

String formatting (or string interpolation) allows us to insert variables or expressions into predefined text.

- Dynamically build messages
- Create readable output
- Format numbers and dates

It is essential for logging, reporting, and user messages.

Basic Example

String interpolation example:

```
custom_string = "String formatting"
print(f"{custom_string} is a powerful technique")
```

It replaces placeholders with real values at runtime.

Methods for Formatting

Python provides three main approaches:

- **str.format()**
- **f-strings** (formatted string literals)
- **Template strings**

Each has advantages depending on the use case.

Positional Formatting

Placeholders {} are replaced in order using `.format()`.

```
print("Machine learning provides {} the ability to learn {}"
      .format("systems", "automatically"))
```

Values are inserted according to their position.

Using Variables

Both the template and values can be variables.

```
my_string = "{} rely on {} datasets"
method = "Supervised algorithms"
condition = "labeled"

print(my_string.format(method, condition))
```

Improves flexibility and reusability.

Reordering Values

Use indices inside placeholders.

```
print("{2} has a friend called {0} and a sister called {1}"
      .format("Betty", "Linda", "Daisy"))
```

Indices allow reordering arguments.

Named Placeholders

Named placeholders improve readability.

```
tool = "Unsupervised algorithms"
goal = "patterns"

print("{title} try to find {aim}"
      .format(title=tool, aim=goal))
```

More descriptive than positional formatting.

Using Dictionaries

```
my_methods = {"tool": "Unsupervised algorithms",
              "goal": "patterns"}

print("{data[tool]} find {data[goal]}"
      .format(data=my_methods))
```

You can access dictionary values directly.

Format Specifiers

Control numeric formatting using :specifier.

```
print("Only {0:.2f}% analyzed"
      .format(0.5155675))
```

`.2f` limits decimals to 2.

Formatting datetime

```
from datetime import datetime

print("Today's date is {:%Y-%m-%d}"
      .format(datetime.now()))
```

Use datetime format codes inside braces.

Formatted String Literals (f-strings)

Add prefix `f` before the string.

```
way = "code"
method = "learning Python faster"

print(f"Practicing how to {way} is the best method for {method}")
```

More readable and concise than `format()`.

Type Conversion

Allowed conversions:

- !s → string
- !r → representation
- !a → ASCII representation

```
name = "Python"
print(f"{name!r}")
```

Format Specifiers

```
number = 90.41890417471841
print(f"{number:.2f}%")
```

Same formatting rules as `format()`.

Datetime in f-strings

```
from datetime import datetime
today = datetime.now()

print(f"{today:%B %d, %Y}")
```

Readable and compact.

Dictionary Lookups

```
family = {"dad": "John"}
print(f"Is your dad {family['dad']}?")
```

Quotes are required inside f-strings.

Escape Sequences

Use backslash to escape characters.

```
my_string = "My dad is called \"John\""
```

Avoid backslashes inside f-string expressions.

Inline Operations

Expressions can be evaluated inside braces.

```
a = 4
b = 7
print(f"{a} * {b} = {a*b}")
```

This is a major advantage of f-strings.

Calling Functions

```
def add(a, b):
    return a + b

print(f"Result: {add(10, 20)}")
```

Functions can be executed inline.

Template Strings

- Simpler syntax
- Slower than f-strings
- Limited formatting capabilities

Useful for external or user-provided templates.

Basic Template Syntax

```
from string import Template

my_string = Template("Data science is $adjective")
print(my_string.substitute(adjective="powerful"))
```

Uses $identifier placeholders.

Multiple Substitutions

```
my_string = Template("$title is $description")
my_string.substitute(title="Python",
                     description="versatile")
```

Multiple placeholders allowed.

—

26≣ Escaping Dollar Signs

“‘latex

Escaping $

```
Template("Price: $$ $value")
```

Use $$ to display a literal dollar sign.

Handling Missing Keys

```
try:
    my_string.substitute(data)
except KeyError:
    print("Missing information")
```

`substitute()` raises an error if missing.

Safe Substitution

```
my_string.safe_substitute(data)
```

Does not raise errors — leaves placeholders unchanged.

Which Should I Use?

f-strings → Best choice (Python 3.6+)
str.format() → Good compatibility
Template → External/user-provided strings

In modern Python, prefer f-strings.