Searching, Indexing & Method Chaining

AiCE Software Engineering Foundations — Python Programming

This closing lesson on strings covers how to check whether text exists inside other text, how to reach into a string and pull out individual characters or portions of it, and how to combine several string methods into a single, elegant line of code.

1. The in Keyword — Checking If a Substring Exists

The in keyword checks whether one string exists as a substring within another, returning True if found and False if not — this is called a membership test:

```
slogan = "Restoring Africa's Global Leadership Through
Technology"

print("Africa" in slogan)
print("Europe" in slogan)
print("technology" in slogan)    # case-sensitive!
```

This produces:

```
True
False
False
```

The in keyword returns a boolean value — True or False. Notice that 'technology' (lowercase t) returns False even though 'Technology' (uppercase T) exists in the slogan — in is case-sensitive, just like everything else in Python. You can also use not in to check for the absence of a substring:

```
print("Europe" not in slogan)   # True – Europe is not in the
slogan
```

2. String Indexing and Slicing

Because a string is an ordered sequence of characters, each character has a position, called an index. Python uses zero-based indexing — the first character sits at index 0, the second at index 1, and so on.

```
country = "Nigeria"

# Accessing individual characters by index
print(country[0])    # N — first character
print(country[6])    # a — last character

# Negative indexing — counting from the end
print(country[-1])   # a — last character
```

Slicing extracts a portion of a string:

```
print(country[0:3])  # Nig — characters at index 0, 1, 2
print(country[3:])   # eria — from index 3 to the end
print(country[:3])   # Nig — from the start to index 2
```

Slicing syntax is string[start:end] — it includes the character at the start index but excludes the character at the end index. This is called exclusive end slicing, and it's consistent across Python's data structures.

Syntax	Meaning	Example with 'Nigeria'
string[0]	First character	N
string[-1]	Last character	a
string[0:3]	Characters at index 0, 1, 2	Nig
string[3:]	From index 3 to the end	eria
string[:3]	From the start to index 2	Nig

3. .find() — Locating a Substring

.find() searches for a substring and returns the index position where it first appears. If the substring is not found, it returns -1:

```
sentence = "African developers are building the future of
Africa."

print(sentence.find("Africa"))
print(sentence.find("Europe"))

# Practical use: check if something exists before using it
if sentence.find("Africa") != -1:
    print("The word Africa was found in the sentence.")
```

This produces:

```
0
-1
The word Africa was found in the sentence.
```

> **in vs. .find()**
> *Use in when you only need to know whether a substring exists (True/False). Use .find() when you also need to know where it is (the index). Both are useful in different situations.*

4. Combining Methods — Method Chaining

Because string methods return new strings, you can chain multiple methods together in a single line — the output of one becomes the input of the next:

```
# Raw user input simulation — messy data
raw_input = "   AMARA diallo   "

# Clean it in one chained line
```

```
clean_name = raw_input.strip().title()
print(f"Cleaned: '{clean_name}'")

# Chain three methods
messy = "   *** african tech leaders ***   "
clean = messy.strip().replace("*", "").strip().title()
print(clean)
```

This produces:

```
Cleaned: 'Amara Diallo'
African Tech Leaders
```

.strip().title() first strips the whitespace, then applies title case to the result. You can chain as many methods as needed, reading left to right — each one processes the output of the previous.

5. Bringing It All Together — The AiCE Student Data Cleaner

A realistic program combining nearly every method from this unit — cleaning messy registration data exactly as it might arrive from a real form submission:

```
# Raw data as it might arrive from a form submission
raw_name = "   FATIMA al-hassan   "
raw_email = "   Fatima.AlHassan@GMAIL.com   "
raw_phone = "+249-91-234-5678"

# Clean and format each field
clean_name = raw_name.strip().title()
clean_email = raw_email.strip().lower()
clean_phone = raw_phone.replace("-", "")

# Display the cleaned profile
print(f"Name  : {clean_name}")
print(f"Email : {clean_email}")
```

```
print(f"Phone : {clean_phone}")

# Validation checks
print(f"Name length valid (>3): {len(clean_name) > 3}")
print(f"Email contains @: {"@" in clean_email}")
```

The output turns messy raw input into clean, consistently formatted data — exactly what happens in real registration systems, user account flows, and data pipelines. The validation checks at the bottom show how string methods feed directly into logic, a pattern you'll use constantly once conditionals are introduced.

Key Takeaways

in tells you whether a substring exists; .find() tells you where. Indexing (string[i]) reaches a single character, while slicing (string[i:j]) extracts a range — always inclusive of the start, exclusive of the end. Method chaining lets you combine several transformations, like .strip().title(), into one clean, readable line. With strings and their methods fully covered, the next lesson moves from text to numbers — Python's arithmetic operators and the rules of calculation.