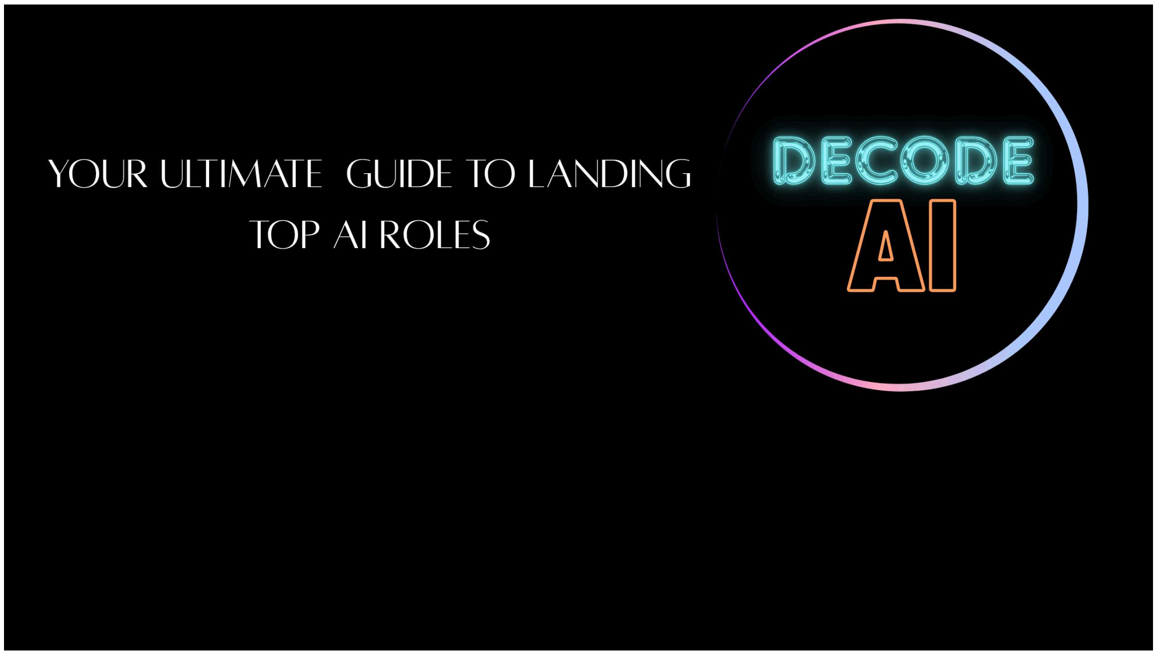

Strings in Python

Textual data in Python is handled with `str` objects, or strings. Strings are **immutable sequences** of **Unicode code points**.

Strings are also containers, because they hold a group of items (characters).

String literals are written in a variety of ways:

```python
# Single quotes
s1 = 'allows embedded "double" quotes'

# Double quotes
s2 = "allows embedded 'single' quotes"

# Triple single quotes (for multi-line or embedded quotes)
s3 = '''Three single quotes'''

# Triple double quotes (also useful for docstrings)
s4 = """Three double quotes"""
```

Properties of string

Property	String supports?	Example
Sequence	✅ Yes	`s[0]`, `s[1:4]`, `for ch in s`
Container	✅ Yes	`'e' in s`, `len(s)`
Mutable	❌ No	Strings are **immutable**

Property	String supports?	Example
Ordered	✅ Yes	Preserves character order
Duplicates	✅ Yes	Allows repeated characters

```
In [11]: # Single word
         'hello'

Out[11]: 'hello'
```

```
In [12]: # Entire phrase
         'This is also a string'

Out[12]: 'This is also a string'
```

```
In [13]: # We can also use double quote
         "String built with double quotes"

Out[13]: 'String built with double quotes'
```

```
In [14]: # Be careful with quotes!
         ' I'm using single quotes, but will create an error'

  Cell In[14], line 2
    ' I'm using single quotes, but will create an error'
                                                       ^
SyntaxError: unterminated string literal (detected at line 2)
```

```
In [15]: "Now I'm ready to use the single quotes inside a string!"

Out[15]: "Now I'm ready to use the single quotes inside a string!"
```

Printing a String

```
In [16]: # Printing a String - We can simply declare a string
         'Hello World'

Out[16]: 'Hello World'
```

```
In [17]: # Printing a String - note that we can't output multiple strings this way
         'Hello World 1'
         'Hello World 2'

Out[17]: 'Hello World 2'
```

```
In [18]: # print len
         len('Hello World')

Out[18]: 11
```

Indexing in string

In [19]:
```
# Assign s as a string
s = 'Hello World'
```

In [20]:
```
#Check
s
```

Out[20]:
```
'Hello World'
```

In [21]:
```
# Print the object
print(s)
```

```
Hello World
```

In [22]:
```
# Show first element (in this case a letter)
s[0]
```

Out[22]:
```
'H'
```

In [23]:
```
s[2]
```

Out[23]:
```
'l'
```

In [24]:
```
# Grab everything past the first term all the way to the length of s which is len(s
s[1:]
```

Out[24]:
```
'ello World'
```

In [25]:
```
# Note that there is no change to the original s
s
```

Out[25]:
```
'Hello World'
```

In [26]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[26]:
```
'Hel'
```

In [27]:
```
#Everything
s[:]
```

Out[27]:
```
'Hello World'
```

In [28]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[28]:
```
'd'
```

In [29]:
```
# Grab everything but the last letter
s[:-1]
```

Out[29]:
```
'Hello Worl'
```

```
In [30]: # Grab everything, but go in steps size of 1
         s[::1]

Out[30]: 'Hello World'

In [31]: # Grab everything, but go in step sizes of 2
         s[::2]

Out[31]: 'HloWrd'

In [32]: # We can use this to print a string backwards
         s[::-1]

Out[32]: 'dlroW olleH'
```

String Immutability

```
In [68]: s

Out[68]: 'hello'

In [35]: # Let's try to change the first letter to 'x'
         s[0] = 'x'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 2
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment

In [36]: # Concatenate strings!
         s + ' concatenate me!'

Out[36]: 'Hello World concatenate me!'

In [69]: # We can reassign s completely though!
         s = s + ' concatenate me!'

In [92]: s

Out[92]: 'hello'

In [39]: letter = 'z'

In [40]: letter*10

Out[40]: 'zzzzzzzzzz'
```

Basic Built-in String methods

```
In [41]: s
```

```
Out[41]: 'Hello World concatenate me!'
```

```
In [42]: # Upper Case a string
         s.upper()
```

```
Out[42]: 'HELLO WORLD CONCATENATE ME!'
```

```
In [43]: # Lower case
         s.lower()
```

```
Out[43]: 'hello world concatenate me!'
```

```
In [44]: # Split a string by blank space (this is the default)
         s.split()
```

```
Out[44]: ['Hello', 'World', 'concatenate', 'me!']
```

```
In [104…  # Split by a specific element (doesn't include the element that was split on)
          s = "hello, how are you?"
          s.split(',')
```

```
Out[104…  ['hello', ' how are you?']
```

```
In [105…  'ab c\n\nde fg\rkl\r\n'.splitlines()
```

```
Out[105…  ['ab c', '', 'de fg', 'kl']
```

```
In [98]: " ".join([" hi","good morning !"])
```

```
Out[98]: ' hi good morning !'
```

```
In [99]: "Hello world".lower()
```

```
Out[99]: 'hello world'
```

```
In [100…  " Hello World! ".lstrip()
```

```
Out[100…  'Hello World! '
```

```
In [103…  " Hello World! ".rstrip()
```

```
Out[103…  ' Hello World!'
```

```
In [102…  " Hello World! ".replace("Hello","hi")
```

```
Out[102…  ' hi World! '
```

Print Formatting

In [77]:
```
"The sum of 1 + 2 is {0}".format(1+2)
```
Out[77]:
```
'The sum of 1 + 2 is 3'
```

In [78]:
```
f"The sum of 1+2 is {1+2}"
```
Out[78]:
```
'The sum of 1+2 is 3'
```

Location and Counting

In [71]:
```
s = 'hello concatenate me!'
```

In [74]:
```
s.count('concatenate')
```
Out[74]:
```
1
```

In [75]:
```
s.find('concatenate')
```
Out[75]:
```
6
```

In [76]:
```
'concatenate' in s
```
Out[76]:
```
True
```

is check methods

In [80]:
```
s = 'hello'
```

In [81]:
```
s.isalnum() # chars are alphanumeric and there is at least one character
```
Out[81]:
```
True
```

In [53]:
```
s.isalpha()
```
Out[53]:
```
True
```

In [82]:
```
s.isascii()
```
Out[82]:
```
True
```

In [85]:
```
s.isdigit()
```
Out[85]:
```
False
```

In [54]:
```
s.islower()
```
Out[54]:
```
True
```

In [89]:
```
s.isnumeric()
```
Out[89]:
```
False
```

```
In [55]: s.isspace()
```

```
Out[55]: False
```

```
In [56]: s.istitle()
```

```
Out[56]: False
```

```
In [57]: s.isupper()
```

```
Out[57]: False
```

Built-in Reg. Expressions

```
In [59]: s.split('e')
```

```
Out[59]: ['h', 'llo']
```

```
In [60]: s.partition('e')
```

```
Out[60]: ('h', 'e', 'llo')
```

Miscellaneous methods

```
In [66]: s.endswith('o')
```

```
Out[66]: True
```

```
In [65]: #str.encode(encoding='utf-8', errors='strict')

english_text = 'Python is Cool'

encoded = english_text.encode('utf-8')  # Encoding to bytes
print("Encoded:", encoded)

decoded = encoded.decode('utf-8')  # Decoding back to string
print("Decoded:", decoded)

emoji_text = "हेलो 👋🌍" #Unicode
encoded = emoji_text.encode('utf-8')  # Encoding to bytes
print("Encoded:", encoded)

decoded = encoded.decode('utf-8')  # Decoding back to string
print("Decoded:", decoded)
```

```
Encoded: b'Python is Cool'
Decoded: Python is Cool
Encoded: b'\xe0\xa4\xb9\xe0\xa5\x87\xe0\xa4\xb2\xe0\xa5\x8b \xf0\x9f\x91\x8b\xf0\x9f
\x8c\x8d'
Decoded: हेलो 👋🌍
```

```
In [ ]:
```

[Bonus Section] Decoding Internal of Strings

In Python, strings are immutable sequences of Unicode characters. Behind the scenes, their internal representation is optimized for memory efficiency, speed, and Unicode support. Some Important techniques adopted by strings are

Feature	ASCII	Unicode
Full Form	American Standard Code for Information Interchange	Universal Coded Character Set
Bit size	7 bits (128 characters)	Typically uses 8, 16, or 32 bits
Characters supported	English letters, digits, symbols	All major languages, emojis, symbols, scripts
Range	`0 - 127`	`0 - 1,114,111` (code points)
Memory use	Very low	Depends on encoding (UTF-8, UTF-16, etc.)
Encoding Type	Fixed-width	Variable-width (UTF-8), fixed (UTF-32)
Backward Compatible	✅ Yes	✅ Yes (UTF-8 includes ASCII)

1. Flexible String Representation

- Python now uses **compact, memory-efficient internal formats** based on the kind of characters present:

Kind of Characters	Storage Format	Bytes per Char
ASCII only	Latin-1	1 byte
Latin-1 + others	UCS-2	2 bytes
Emojis / CJK	UCS-4	4 bytes

- For example:
  - `s = "abc"` → uses **1 byte per character**
  - `s = "हेलो"` (Hindi) → may use **2 bytes per character**
  - `s = "👋🌍"` (emojis) → may use **4 bytes per character**
- This approach helps **reduce memory usage** and **speed up operations**.

In [7]:
```python
import sys

# for empty string
ascii_str = ""
```

```
print(f"Empty String: {ascii_str}")
print(f"Memory (bytes): {sys.getsizeof(ascii_str)}")

# Hindi String (UCS-2) =>  Only 1 byte per character is needed
ascii_str = "abc"
print(f"ASCII String: {ascii_str}")
print(f"Memory (bytes): {sys.getsizeof(ascii_str)}")

# ASCII Only string =>  This string uses Devanagari characters, requiring 2 bytes p
hindi_str = "हेलो"
print(f"Hindi String: {hindi_str}")
print(f"Memory (bytes): {sys.getsizeof(hindi_str)}")

# Emoji String (UCS-4) => Emojis are outside the Basic Multilingual Plane (BMP), so
emoji_str = "👋🌍"
print(f"Emoji String: {emoji_str}")
print(f"Memory (bytes): {sys.getsizeof(emoji_str)}")
```

```
Empty String:
Memory (bytes): 41
ASCII String: abc
Memory (bytes): 44
Hindi String: हेलो
Memory (bytes): 66
Emoji String: 👋🌍
Memory (bytes): 68
```

2. String Interning

- **Interning** is a mechanism that stores only **one copy** of immutable strings with the same content.
- The single copy of each string is called its **intern**, hence the name *String Interning*.
- Modern programming languages like **Java, Python, PHP, Ruby, Julia**, and many more perform string interning to make their compilers and interpreters more performant.

Advantages of String Interning:

- **Advantage 1: Faster comparison**
  - Interned strings allow for pointer equality using `is` instead of content equality using `==`.
  - Without interning, comparing two strings takes `O(n)` time as every character must be checked.
  - With interning, two equal strings will share the same object reference, so only a pointer comparison is needed.
- **Advantage 2: Reduced memory usage**
  - Instead of creating multiple redundant string objects, Python reuses the same interned object.
  - This helps reduce the memory footprint by avoiding duplication.

In [8]:
```
a = "hello"
b = "hello"

print(a == b)  # True - same content
print(a is b)  # True - same memory (auto interned)
```

```
True
True
```

In [9]:
```
# strings with space or special characters
x = "hello world"
y = "hello world"

print(x == y)  # True - content is same
print(x is y)  # False - different memory (not interned automatically)
```

```
True
False
```

3. Operations Are Optimized

Even though strings are immutable, Python optimizes:

- **Concatenation** using `join()` (avoids `O(n²)` cost of using `+`)
- **Hashing** for dict keys (cached in the string object)
- **Encoding/Decoding** with UTF-8 or other formats

In [2]:
```
# Using + in loops creates a new string object every time due to string immutabilit
words = ["python", "is", "fast"]
result = ""
for word in words:
    result += word + " "
print(result.strip())

# solution - Using " ".join() (Efficient)
```

```
python is fast
```

In [3]:
```
text = "हेलो 👋🌍" #Unicode
encoded = text.encode('utf-8')  # Encoding to bytes
print("Encoded:", encoded)

decoded = encoded.decode('utf-8')  # Decoding back to string
print("Decoded:", decoded)

# UTF-8 is the default encoding and is highly optimized in CPython.
# Operations like encode()/decode() are implemented in fast C code for performance.
```

```
Encoded: b'\xe0\xa4\xb9\xe0\xa5\x87\xe0\xa4\xb2\xe0\xa5\x8b \xf0\x9f\x91\x8b\xf0\x9f
\x8c\x8d'
Decoded: हेलो 👋🌍
```

In [ ]:
```
# Demo
a = "hello"
print(id(a))   # e.g., 140014090012048
```

```
a += " world"
print(id(a))   # Different ID – new object created
```

4. Memory Layout & Overhead

Each string has:

1. A `PyObject_HEAD` (basic metadata)
2. `length`
3. `hash` value (cached after first computation)
4. `kind` – encoding kind (1/2/4-byte characters)
5. `UTF-8` or raw memory buffers

So, even an empty string isn’t exactly "empty"

In [10]:
```
import sys
print(sys.getsizeof(""))  # ~49 bytes (varies by version)
```

```
41
```

📚 References

1. https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
2. https://arpitbhayani.me/blogs/string-interning-python/