Python String - Complete Notes

What is a String?

A string in Python is a **sequence of characters** enclosed in quotes. Strings are **immutable**, meaning they cannot be changed after creation.

Key Characteristics:

- **Immutable**: Cannot be modified after creation
- **Ordered**: Characters have a specific sequence
- **Iterable**: Can loop through characters
- **Indexed**: Access characters by position (0-based indexing)
- **Unicode**: Supports all Unicode characters by default

Creating Strings

```
# Single quotes
str1 = 'Hello World'

# Double quotes
str2 = "Hello World"

# Triple quotes (for multiline strings)
str3 = """This is a
multiline
string"""

str4 = '''Another way to
create multiline
strings'''

# Empty string
empty_str = ""
empty_str2 = str()

# Raw strings (backslashes treated literally)
raw_str = r"C:\Users\name\folder"

# Unicode strings
unicode_str = "Hello     "
```

String Indexing and Slicing

Indexing

```
text = "Python"
#       P y t h o n
#       0 1 2 3 4 5
#      -6-5-4-3-2-1

print(text[0])   # 'P'
print(text[-1])  # 'n'
print(text[2])   # 't'
print(text[-2])  # 'o'
```

Slicing

```
text = "Programming"

# Basic slicing [start:end:step]
print(text[0:4])    # 'Prog'
print(text[4:])     # 'ramming'
print(text[:4])     # 'Prog'
print(text[:])      # 'Programming' (entire string)

# Step parameter
print(text[::2])    # 'Pormin' (every 2nd character)
print(text[::-1])   # 'gnimmargorP' (reverse string)
print(text[1::2])   # 'rgamn'

# Negative indexing in slicing
print(text[-4:-1])  # 'min'
print(text[-4:])    # 'ming'
```

String Operations

1. Concatenation

```
str1 = "Hello"
str2 = "World"

# Using + operator
result = str1 + " " + str2  # "Hello World"

# Using += operator
str1 += " " + str2  # Modifies str1

# Using join() (more efficient for multiple strings)
words = ["Hello", "Beautiful", "World"]
```

```
result = " ".join(words)  # "Hello Beautiful World"

# Using f-strings (Python 3.6+)
name = "Alice"
age = 25
message = f"Hello, my name is {name} and I'm {age} years old"
```

2. Repetition

```
text = "Python"
repeated = text * 3  # "PythonPythonPython"

# Creating separators
separator = "-" * 50  # 50 dashes
```

3. Membership Testing

```
text = "Hello World"

print("Hello" in text)     # True
print("hello" in text)     # False (case sensitive)
print("xyz" not in text)   # True
```

String Methods

1. Case Conversion Methods

```
text = "Hello World"

print(text.upper())        # "HELLO WORLD"
print(text.lower())        # "hello world"
print(text.title())        # "Hello World"
print(text.capitalize())   # "Hello world"
print(text.swapcase())     # "hELLO wORLD"

# Case checking
print(text.isupper())      # False
print(text.islower())      # False
print(text.istitle())      # True
```

2. Whitespace Methods

```
text = "  Hello World  "

print(text.strip())        # "Hello World"
print(text.lstrip())       # "Hello World  "
print(text.rstrip())       # "  Hello World"
```

```
# Strip specific characters
text2 = "...Hello World..."
print(text2.strip('.'))    # "Hello World"

# Center, ljust, rjust
print(text.strip().center(20, '-'))  # "---Hello World----"
print(text.strip().ljust(15))        # "Hello World    "
print(text.strip().rjust(15))        # "    Hello World"
```

3. Search and Replace Methods

```
text = "Hello World Hello"

# Finding substrings
print(text.find("Hello"))      # 0 (first occurrence)
print(text.find("Hello", 1))   # 12 (search from index 1)
print(text.find("xyz"))        # -1 (not found)

print(text.index("Hello"))     # 0 (raises ValueError if not found)
print(text.rfind("Hello"))     # 12 (last occurrence)

# Counting occurrences
print(text.count("Hello"))     # 2
print(text.count("l"))         # 6

# Replace
print(text.replace("Hello", "Hi"))          # "Hi World Hi"
print(text.replace("Hello", "Hi", 1))       # "Hi World Hello" (replace only first)

# Check start/end
print(text.startswith("Hello"))  # True
print(text.endswith("World"))    # False
print(text.endswith("Hello"))    # True
```

4. Split and Join Methods

```
text = "apple,banana,orange"
csv_text = "name,age,city\nJohn,25,NYC\nAlice,30,LA"

# Split
fruits = text.split(',')         # ['apple', 'banana', 'orange']
words = "Hello World".split()    # ['Hello', 'World'] (default: whitespace)
lines = csv_text.split('\n')     # Split by newlines

# Partition
result = "name=value".partition('=')  # ('name', '=', 'value')

# Join
fruits_str = " | ".join(fruits)  # "apple | banana | orange"
```

```
numbers = [str(i) for i in range(5)]
number_str = "".join(numbers)    # "01234"
```

5. Validation Methods

```
# Character type checking
print("123".isdigit())        # True
print("abc".isalpha())        # True
print("abc123".isalnum())     # True
print("   ".isspace())        # True
print("Hello World".isascii()) # True
print("Hello123".isdecimal())  # False

# String content validation
print("".isidentifier())      # False
print("variable_name".isidentifier())  # True
print("123abc".isidentifier())         # False
print("Hello World".isprintable())     # True
```

6. Format Methods

```
name = "Alice"
age = 25
salary = 50000.50

# Old style formatting
old_format = "Name: %s, Age: %d" % (name, age)

# str.format() method
new_format = "Name: {}, Age: {}".format(name, age)
named_format = "Name: {name}, Age: {age}".format(name=name, age=age)
indexed_format = "Name: {0}, Age: {1}".format(name, age)

# f-strings (Python 3.6+) - Recommended
f_string = f"Name: {name}, Age: {age}"
f_string_expr = f"Next year {name} will be {age + 1}"
f_string_format = f"Salary: ${salary:,.2f}"  # $50,000.50

# Advanced formatting
print(f"{name:>10}")      # Right align in 10 chars
print(f"{name:<10}")      # Left align in 10 chars
print(f"{name:^10}")      # Center align in 10 chars
print(f"{age:04d}")       # Zero-padded to 4 digits
print(f"{salary:.1f}")    # One decimal place
```

String Encoding and Decoding

```
text = "Hello 🌍"

# Encode to bytes
utf8_bytes = text.encode('utf-8')
ascii_bytes = text.encode('ascii', errors='ignore')  # Ignores non-ASCII chars

# Decode from bytes
decoded_text = utf8_bytes.decode('utf-8')

# Common encodings
print(text.encode('utf-8'))    # b'Hello \xf0\x9f\x8c\x8d'
print(text.encode('latin1', errors='ignore'))  # Ignores unsupported chars
```

String Comparison

```
str1 = "apple"
str2 = "banana"
str3 = "Apple"

# Lexicographic comparison
print(str1 < str2)    # True
print(str1 == str3)   # False
print(str1.lower() == str3.lower())  # True

# Case-insensitive comparison
def case_insensitive_compare(s1, s2):
    return s1.lower() == s2.lower()

print(case_insensitive_compare("Hello", "HELLO"))  # True
```

Advanced String Operations

1. String Templating

```
from string import Template

# Template strings
template = Template("Hello $name, you have $count messages")
result = template.substitute(name="Alice", count=5)
# "Hello Alice, you have 5 messages"

# Safe substitution (doesn't raise error for missing keys)
result2 = template.safe_substitute(name="Bob")
# "Hello Bob, you have $count messages"
```

2. Regular Expressions

```
import re

text = "Contact: john@example.com or call 123-456-7890"

# Find email
email = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(email)  # ['john@example.com']

# Find phone number
phone = re.findall(r'\d{3}-\d{3}-\d{4}', text)
print(phone)  # ['123-456-7890']

# Replace pattern
clean_text = re.sub(r'\d{3}-\d{3}-\d{4}', '[PHONE]', text)
```

3. String Performance Tips

```
# Efficient string concatenation
words = ["apple", "banana", "cherry"]

# SLOW for large lists
slow_concat = ""
for word in words:
    slow_concat += word + " "

# FAST
fast_concat = " ".join(words)

# Using list comprehension with join
result = " ".join([word.upper() for word in words])
```

String Constants (from string module)

```
import string

print(string.ascii_letters)     # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
print(string.ascii_lowercase)   # 'abcdefghijklmnopqrstuvwxyz'
print(string.ascii_uppercase)   # 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(string.digits)            # '0123456789'
print(string.punctuation)       # '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
print(string.whitespace)        # ' \t\n\r\x0b\x0c'
```

Interview Tricky Questions & Answers

1. String Immutability Gotcha

```
# What happens here?
s = "hello"
s[0] = 'H'  # TypeError: 'str' object does not support item assignment

# Correct way
s = "hello"
s = 'H' + s[1:]  # "Hello"
# Or
s = s.replace('h', 'H', 1)  # "Hello"
```

2. String Interning

```
# What's the output?
a = "hello"
b = "hello"
print(a is b)  # True (strings are interned)

a = "hello world"
b = "hello world"
print(a is b)  # May be True or False (implementation dependent)

# Force no interning
import sys
a = sys.intern("hello")
b = sys.intern("hello")
print(a is b)  # Always True
```

3. Mutable Default Arguments with Strings

```
# This is actually safe with strings (unlike lists)
def append_suffix(text, suffix="default"):
    return text + suffix

result1 = append_suffix("hello", "!")
result2 = append_suffix("world")
# No issues because strings are immutable
```

4. String Formatting Precision

```
# What's the difference?
value = 3.14159

print("%.2f" % value)        # 3.14
print("{:.2f}".format(value)) # 3.14
print(f"{value:.2f}")        # 3.14
```

```python
# Tricky case
print(f"{2.5:.0f}")  # 2 (banker's rounding)
print(f"{3.5:.0f}")  # 4 (banker's rounding)
```

5. String Comparison Gotcha

```python
# What happens here?
print("10" > "9")    # False! (lexicographic comparison)
print("10" > "2")    # False!
print(int("10") > int("9"))  # True

# Solution for numeric string comparison
def numeric_string_compare(s1, s2):
    return int(s1) - int(s2)
```

6. Split Behavior

```python
# Tricky split behavior
text1 = "a,,b"
text2 = "a  b"  # Two spaces
text3 = "  a  b  "

print(text1.split(','))     # ['a', '', 'b']
print(text2.split(' '))     # ['a', '', 'b']
print(text2.split())        # ['a', 'b'] (default splits on any whitespace)
print(text3.split())        # ['a', 'b'] (strips and splits)

# Difference between split() and split(' ')
print("a b  c".split())     # ['a', 'b', 'c']
print("a b  c".split(' '))  # ['a', 'b', '', 'c']
```

7. String Escape Sequences

```python
# What's the output?
print(len("hello\nworld"))   # 11 (\n is one character)
print(len(r"hello\nworld"))  # 12 (raw string, \n is two characters)

print("C:\new\folder")       # C:
                            # ew\folder (\n is newline)
print(r"C:\new\folder")      # C:\new\folder

# Unicode escapes
print("\u0041")              # A
print("\U00000041")          # A
```

8. String Multiplication with Lists

```python
# What's the output?
result = "abc" * 3
```

```python
print(result)  # "abcabcabc"

# But be careful with this pattern
def create_matrix(rows, cols):
    # WRONG - all rows reference the same list!
    matrix = [["0"] * cols] * rows
    matrix[0][0] = "X"
    return matrix

# Correct way
def create_matrix_correct(rows, cols):
    matrix = [["0"] * cols for _ in range(rows)]
    return matrix
```

9. String Methods Return New Objects

```python
# What's the value of original_string?
original_string = "  hello world  "
trimmed = original_string.strip()
upper_case = original_string.upper()

print(original_string)  # "  hello world  " (unchanged!)
print(trimmed)          # "hello world"
print(upper_case)       # "  HELLO WORLD  "

# Strings are immutable - methods return new strings
```

10. Boolean Context of Strings

```python
# What's the output?
strings = ["", "0", "False", "None", "hello", " "]

for s in strings:
    if s:
        print(f"'{s}' is truthy")
    else:
        print(f"'{s}' is falsy")

# Output:
# '' is falsy (only empty string is falsy)
# '0' is truthy
# 'False' is truthy
# 'None' is truthy
# 'hello' is truthy
# ' ' is truthy (space is not empty!)
```

11. String Addition vs Join Performance

```
import time

# Inefficient for many strings
def slow_concat(strings):
    result = ""
    for s in strings:
        result += s  # Creates new string object each time!
    return result

# Efficient
def fast_concat(strings):
    return "".join(strings)

# For small numbers, += might be optimized by CPython
# For large numbers, join() is always faster
```

12. Encoding/Decoding Traps

```
# What happens here?
text = "café"  # Contains é
try:
    ascii_bytes = text.encode('ascii')  # UnicodeEncodeError
except UnicodeEncodeError as e:
    print(f"Error: {e}")

# Safe encoding
ascii_safe = text.encode('ascii', errors='ignore')  # b'caf'
ascii_replace = text.encode('ascii', errors='replace')  # b'caf?'

# Decoding
bytes_data = b'\xff\xfe'
try:
    text = bytes_data.decode('utf-8')  # UnicodeDecodeError
except UnicodeDecodeError:
    text = bytes_data.decode('utf-8', errors='ignore')
```

Best Practices

1. **Use f-strings for formatting** (Python 3.6+): Most readable and efficient
2. **Use join() for concatenating many strings**: More efficient than repeated +
3. **Use raw strings for regex patterns**: `r"pattern"` avoids double escaping
4. **Use str.casefold() for case-insensitive comparison**: Better than lower() for Unicode
5. **Use string methods instead of regular expressions**: When simple string methods suffice

6. **Be explicit with encoding**: Always specify encoding when reading/writing files
7. **Use string constants**: `string.ascii_letters` instead of hardcoding
8. **Cache compiled regex patterns**: If using the same pattern multiple times

---

Time Complexity

Operation	Time Complexity	Notes
Access by index	O(1)	Direct access
Slicing	O(k)	k = length of slice
Concatenation (+)	O(n + m)	n, m = lengths of strings
find/index	O(n*m)	n = text length, m = pattern length
replace	O(n*m)	Worst case
split	O(n)	n = string length
join	O(n)	n = total length of result

Memory Considerations

- **String interning**: Python automatically interns some strings for efficiency
- **Immutability**: Each string operation creates a new object
- **Large strings**: Be careful with operations that create many temporary strings
- **Use generators**: For processing large amounts of text data